MASTER EN SCIENCES INFORMATIQUES

Développement d'un module générique de représentation d'applications et d'objets réseaux au sein du produit OpenMaster

Mihy, Steve

Award date:
1999

Facultés Universitaires Notre-Dame de la Paix - Namur

Développement d'un module générique de représentation d'applications et d'objets réseaux au sein du produit *OpenMaster*

Steve Mihy

Mémoire présenté en vue de l'obtention du grade de
Maître en Informatique

Année académique 1998-1999

Développement d'un module générique de représentation d'applications et d'objets réseaux au sein du produit *OpenMaster*

-Résumé-

Ce mémoire s'inscrit dans le cadre du stage que nous avons effectué à Paris chez Bull et plus précisément au sein de la division BullSoft.

Notre travail s'articule autour de quatre chapitres qui marquent une progression dans le développement du *bean WebTree* au sein de l'application *OpenMaster*.

Après une brève description d'*OpenMaster*, notre travail s'attache dans le premier chapitre à examiner les besoins en rapport avec le *WebTree* existant dans les différents projets de développement. Nous passons ensuite au deuxième chapitre à une description des différents concepts clés du *WebTree* ainsi que de l'interface utilisateur de celui-ci.

Le troisième chapitre aborde un point de vue plus technique du *WebTree*. Il présente en effet les différentes technologies que nous avons étudiées afin de déterminer lesquelles étaient utiles à notre application.

Enfin, le dernier chapitre s'attache au développement proprement dit de l'application. Outre notre environnement de travail, nous y décrivons comment nous avons pu générer automatiquement une partie du code et quels sont les problèmes majeurs rencontrés durant le développement.

Development of a generic module to represent network applications and objects within the *OpenMaster* product

-Summary-

This thesis lies within the scope of the training course which we carried out in Paris at Bull and more precisely within the BullSoft division.

Our work is articulated around four chapters which mark a progression in the development of the *bean WebTree* within the *OpenMaster* application.

After a short description of *OpenMaster*, our work sticks in the first chapter to examine the existing requirements in connection with the *WebTree* in the various development projects. We pass then in the second chapter to a description of the various key concepts of *WebTree* as well as the user interface for this one.

The third chapter approaches a more technical point of view of *WebTree*. It indeed presents various technologies which we studied in order to determine which were useful to our application.

Finally, the last chapter sticks to the development itself of the application. In addition to our environment of work, we describe there how we could generate part of the code automatically and which are the major problems encountered during the development.

Remerciements

Au terme de cette année de stage et de rédaction, je voudrais remercier toutes les personnes qui ont collaboré, de près ou de loin, à la réalisation de ce mémoire.

Je remercie tout d'abord Monsieur Ramaekers, mon promoteur, ainsi que son assistant Aimé Kassa, dont les remarques et les conseils m'ont été d'une aide précieuse et efficace.

J'adresse également mes remerciements à toutes les personnes qui m'ont accueilli chez Bull, notamment Monsieur Leygues pour m'avoir accueilli dans son équipe et Monsieur Boukobza, avec qui j'ai beaucoup travaillé, pour ses conseils et ses remarques concernant ce mémoire. Je tiens également à remercier Pierre Andrei, Philippe Baudry et Hoan Bui-Xuan pour leur bonne volonté à répondre à mes questions durant le stage.

Je remercie également Bruno pour tous ses conseils durant le stage et ses aides précieuses en LaTeX durant la rédaction, mais aussi et surtout pour les bons moments de détente à Paris, ainsi que tous les copains de troisième maîtrise qui m'ont soutenu d'une manière ou d'une autre.

Je remercie tout spécialement Anne-France pour sa patience, son soutien, son orthographe et ses nombreuses relectures.

Enfin, un grand merci à toute ma famille pour ses encouragements durant cette période.

Table des matières

Table des figures

Introduction

Avant toute chose, il nous faut expliciter le propos de ce mémoire ainsi que le cadre dans lequel il a été réalisé. C'est l'objet de ces quelques pages.

Ce mémoire s'inscrit dans le cadre du développement d'une application, le *WebTree*, au sein du produit *OpenMaster* développé par BullSoft, une division du groupe Bull dans laquelle nous avons effectué notre stage. Il existe, au sein d'*OpenMaster*, différents projets de développement. Le *WebTree* a été commandité principalement par l'un d'entre eux : le projet *Application GOVernor*, ou *AGOV*.

Le premier chapitre décrira brièvement le produit *OpenMaster* afin d'expliquer le cadre d'existence de l'application à développer. Cette description effectuée, nous passerons à la phase d'*ingénierie des besoins*. Nous réaliserons, durant cette phase, non seulement une description du projet *AGOV* et de ses besoins, mais nous présenterons également des besoins spécifiques à d'autres équipes ainsi que des besoins transversaux à *OpenMaster*. Par *transversal*, on entendra *commun à tous les développements réalisés au sein de l'application OpenMaster*.

Les différents besoins étant définis, nous passerons à une phase de spécification de l'application à développer : le *WebTree*. Ce sera l'objet du deuxième chapitre. Nous décrirons alors les différents concepts existant dans le *WebTree*. A cette fin, nous présenterons aussi *UML*[1] que nous avons utilisé afin de réaliser une partie de l'analyse du *WebTree*. Nous y décrirons également l'interface que nous avons conçue afin de répondre le mieux possible aux différents besoins. Nous n'entrerons pas durant ce chapitre dans des détails techniques d'implémentation ou d'utilisation de l'une ou l'autre technologie. Nous essayerons simplement de présenter l'application du point de vue de l'utilisateur et de l'interface mise à sa disposition.

Le troisième chapitre sera, quant à lui, consacré à présenter les différentes technologies utilisables pour réaliser l'application. En effet, de nombreuses possibilités s'offraient à nous aussi bien du point de vue du développement de l'application (de l'analyse au langage utilisé) qu'au niveau des formats de modélisation et de représentation des données. Nous décrirons donc chacune des technologies que nous avons envisagé d'utiliser avec ses avantages et ses inconvénients. Nous terminerons ce chapitre en expliquant les raisons qui ont motivé l'utilisation de certaines possibilités et l'abandon d'autres.

[1] UML : *Unified Modeling Language*.

Dans le chapitre quatre, nous commencerons par décrire l'environnement de travail dans avons évolué durant notre stage. Nous aborderons ensuite l'analyse et le développement proprement dits de l'application. Durant tout le processus de développement d'une application, l'équipe d'analystes et de développeurs est confrontée à différentes contraintes. Outre l'analyse et le développement, ce chapitre décrira également les différents problèmes que nous avons rencontrés, et la marge de manoeuvre que nous avions dans le développement.

En guise de conclusion, nous essayerons de décrire au mieux les apports du travail d'équipe effectué durant le stage et du travail individuel lié à la rédaction de ce mémoire.

Avant de clôturer cette introduction, il nous reste à préciser quelques prérequis. En effet, bien que nous essayerons tout au long de ce travail d'être compréhensible par un maximum de personnes, certaines notions peuvent néanmoins être nécessaires afin de pouvoir comprendre certains points abordés. *OpenMaster* étant une application permettant l'administration de parcs informatiques, il peut être intéressant d'avoir quelques notions dans ce domaine. De plus, certaines connaissances en analyse et en modélisation orientées objets peuvent également aider à la bonne compréhension de ce mémoire. Enfin, nous considérerons également le langage Java comme connu, non au niveau de la programmation dans ce langage, mais bien au niveau de ses concepts.

Chapitre 1

OpenMaster et ses besoins

Tout développement d'application est réalisé afin de répondre à des besoins, dans quelque domaine que ce soit. Cela peut aller du particulier qui, chez lui, développe un petit système pour gérer ses cassettes vidéos, livres, etc. jusqu'au développement d'un système d'exploitation, application minimale nécessaire sur tout ordinateur. Pour être à même de comprendre les besoins nécessitant un développement, il faut connaître le contexte dans lequel ces besoins existent. Dans certains cas, le contexte est relativement simple et connu de tous, dans d'autres, il mérite une explication complète et détaillée. Une fois le contexte expliqué, on peut alors passer à l'expression des besoins.

Le premier point de ce chapitre décrira brièvement *OpenMaster*. Le sujet de ce mémoire n'étant pas *OpenMaster* en soi, mais bien le développement en son sein de l'application *WebTree*, nous ne décrirons pas en détail tout le produit. Nous essayerons simplement d'en avoir une vue générale en montrant son utilité. Néanmoins, nous nous attarderons parfois sur certains points ayant un rapport direct avec le développement du *WebTree*, un de ces points étant la présentation du commanditaire, le projet *AGOV*.

Après cette description, nous pourrons présenter brièvement les différents desiderata du projet *AGOV* ainsi que ceux provenant d'autres projets existant dans *OpenMaster*.

1.1 *OpenMaster*

La première chose à dire à propos d'*OpenMaster* est qu'il ne s'agit pas d'une application *unique*, mais bien d'un ensemble d'applications *distinctes* les unes des autres. C'est une des raisons pour lesquelles on compte plusieurs projets au sein du développement d'*OpenMaster*, l'un d'entre eux étant le projet *AGOV*. Il existe évidemment d'autres raisons purement organisationnelles. Nous commencerons par décrire la solution *OpenMaster* puis nous présenterons le projet *AGOV*.

1.1.1 Le produit en soi

Des entreprises nationales et multinationales de plus en plus nombreuses déploient des systèmes distribués adaptés à leur organisation interne. La réussite de la mise en oeuvre de tels systèmes dépend de la capacité de l'entreprise à :

1. *maîtriser la complexité du réseau* : en effet, la diversité géographique d'un nombre sans cesse croissant de ressources pose, de façon accrue, la question de la gestion des interconnexions. En ajoutant à cela l'augmentation incessante du nombre d'utilisateurs connectés, du débit nécessaire et du nombre d'applications distribuées, nous obtenons un réseau si complexe que sa gestion est un véritable défi quotidien pour l'entreprise ;

2. *suivre et réduire les coûts* d'exploitation et d'administration de son parc informatique ;

3. *réduire les menaces pesant sur les informations* : la sécurité à mettre en place pour garantir la confidentialité et la pérennité des informations existant sur le réseau revêt également une importance considérable. Qui n'a jamais entendu parler de piratages informatiques, espionnage industriel, etc. ?

4. *résoudre les problèmes dus à l'hétérogénéité des ressources informatiques* : le déploiement dans l'entreprise de systèmes différents, par diverses organisations, conduit très souvent à des architectures hétérogènes contenant des réseaux locaux, des PC's, des serveurs Unix et NT, des équipements de télécommunication, etc. Cette situation a, dans la plupart des cas, abouti à des coûts de fonctionnement si élevés que les avantages escomptés ont été en totalité anéantis.

Il est donc indispensable pour l'administrateur de tels parcs informatiques de gérer ces contraintes dans la plus grande transparence. On comprendra aisément l'intérêt d'avoir un système d'administration intégré capable de connaître et d'interpréter le comportement complexe d'équipements qui n'obéissent pas aux mêmes règles et ne possèdent pas le même langage.

OpenMaster est une suite logicielle intégrée dédiée à l'administration des systèmes et réseaux d'un système d'information. *OpenMaster* présente une architecture ouverte, orientée objets, conçue spécialement pour des environnements multiprotocolaires. Elle permet en effet d'administrer tout matériel supportant un protocole d'administration tel que SNMP[1] (IETF)[2], CMIP[3] (ISO[4]), DSAC[5] (BULL), SNA[6] (IBM), etc. *OpenMaster* est multiplates-formes et est disponible sur Windows NT, AIX, SUN, SGI, HP, etc.

[1] SNMP : *Simple Network Management Protocol.*
[2] IETF : *Internet Engineering Task Force.*
[3] CMIP : *Common Management Information Protocol.*
[4] ISO : *International Standards Organization.*
[5] DSAC : *Distributed Systems Administration and Control.*
[6] SNA : *Systems Network Architecture.*

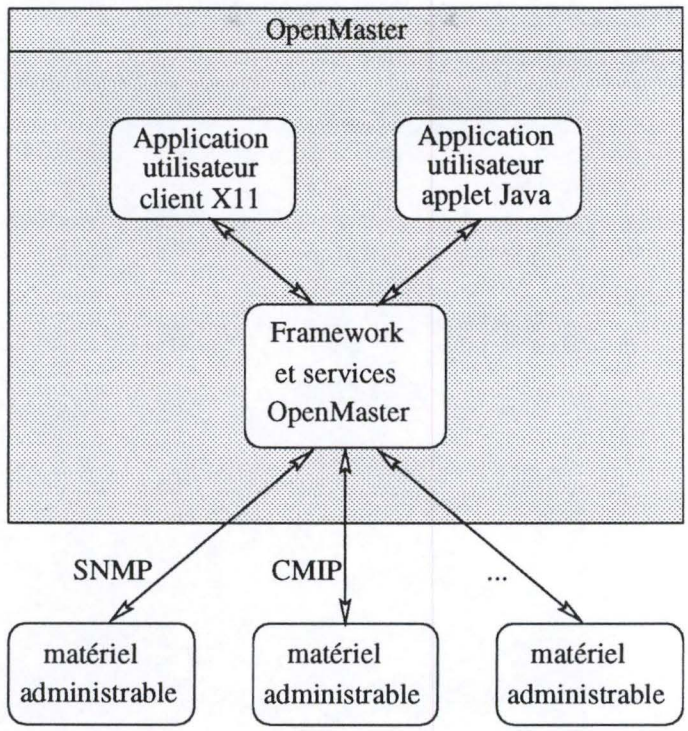


FIG. 1.1 – L'architecture d'*OpenMaster*

OpenMaster est un système ouvert dans la mesure où il permet l'adaptation à de nouveaux types de matériel à administrer, et le développement spécifique de serveurs métiers et d'applications d'administration (*management applications*).

Avec l'explosion d'Internet, le besoin de connexion via le *World Wide Web* entre les plates-formes d'administration *OpenMaster* et les applications utilisateurs ne fait que s'accroître. La nouvelle version d'*OpenMaster* sortie depuis le mois de septembre 1998 propose des applications utilisateurs sous la forme d'une applet Java, le *WebTop*, dans le navigateur Internet Explorer de Microsoft.

La figure 1.1 ci-dessus présente l'architecture d'*OpenMaster*. Nous y remarquons principalement le *framework*, élément central du produit, avec les différents services. Les applications d'*OpenMaster*, qu'il s'agisse d'applications *X11*[7] ou d'applications tournant dans l'applet Java, se connectent au *framework* et aux différents services afin d'obtenir les diverses informations à présenter à l'utilisateur. Ces applications ne sont pas forcément (voire même presque jamais pour l'applet Java) exécutées sur l'ordinateur qui joue le rôle de serveur *OpenMaster*. Les services, quant à eux, reçoivent leurs informations des objets administrés grâce aux protocoles SNMP, CMIP et autres. Si un nouvel objet doit être administré, il suffit qu'il utilise un des nombreux protocoles supportés par *OpenMaster*.

[7] X11 : système de fenêtrage pour les systèmes d'exploitation de type Unix.

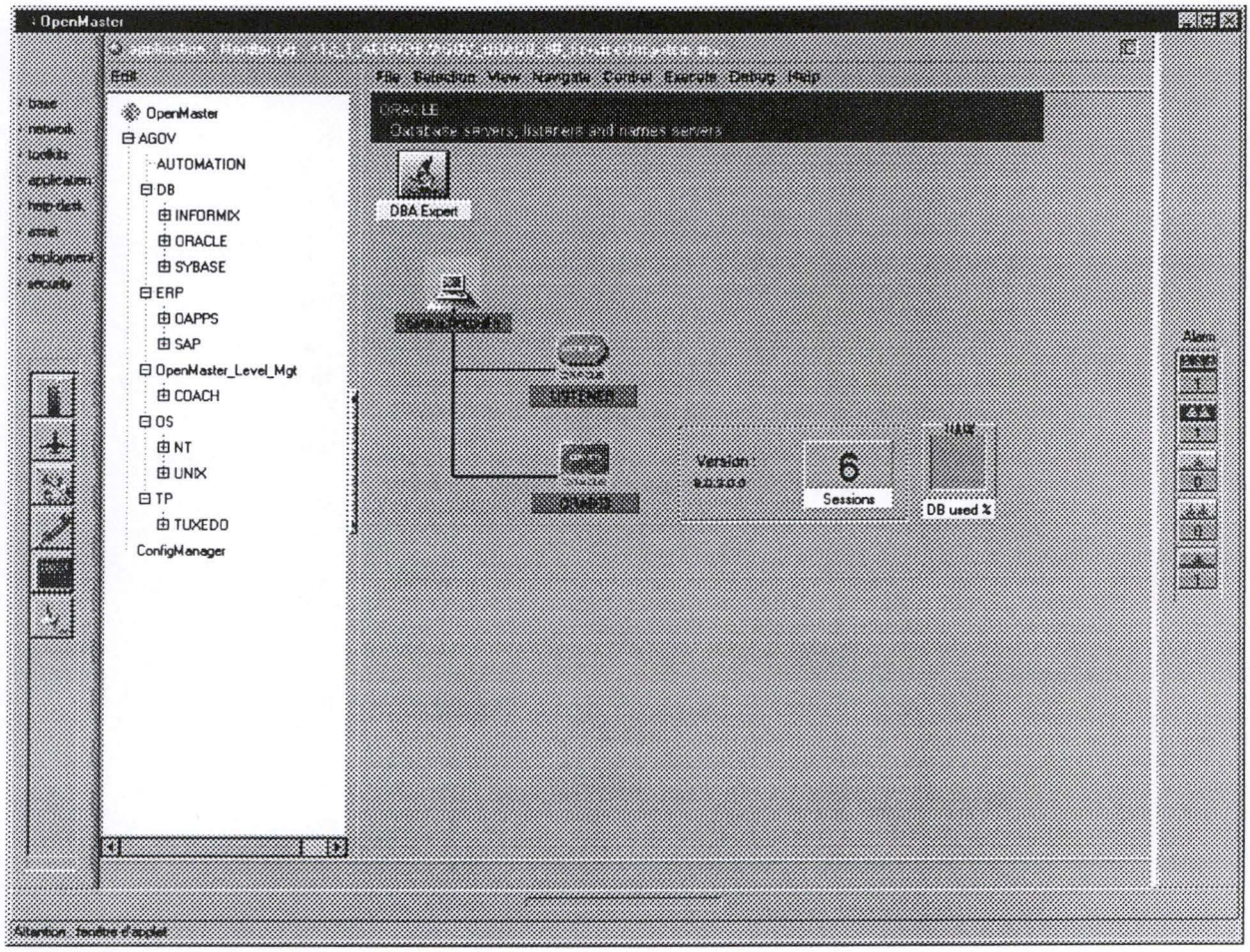


FIG. 1.2 – Le *WebTop*

Il est intéressant à ce niveau de souligner la diversité des objets administrés. En effet, on retrouve parmi ceux-ci non seulement des ordinateurs, mais également des applications. De même, parmi les ordinateurs, nous pouvons distinguer de nombreux matériels distincts. Nous avons bien entendu des ordinateurs personnels, mais également des systèmes plus spécifiques, tels que des *routeurs*, *hubs*, *switchs*, etc. D'un autre côté, au niveau des applications, il s'agira aussi bien de systèmes d'exploitation, d'applications bureautiques, de bases de données que d'applications *ERP*[8].

Le *WebTop*

Comme nous le verrons dans la suite de ce mémoire, il sera souvent question de l'interface *WWW* et donc, plus précisément, de l'applet Java *WebTop*. Celle-ci est présentée par la figure 1.2. Nous pouvons remarquer, à gauche de l'applet, la liste des différents domaines

[8]ERP : *Enterprise Resource Planning.*

(*base*, *network*, *toolkit*, etc.) avec, en dessous, les icônes des différentes applications appartenant au domaine sélectionné. Le centre de l'applet est, quant à lui, réservé à l'affichage des applications en cours d'exécution. Enfin, à droite de l'applet, nous avons une suite d'icônes permettant d'afficher les alarmes survenues.

Examinons un peu plus en détail la partie située au centre de l'applet. Nous y voyons à gauche, sous forme d'arborescence, une application présentant l'ensemble des domaines *AGOV*, et à droite l'application *Monitor*. Nous ne décrirons pas cette dernière, ceci n'étant pas l'objet de ce mémoire.

Les différents domaines pouvant être présentés à l'utilisateur varient en fonction des modules que l'entreprise a achetés. Durant l'exécution de l'applet, l'utilisateur peut passer d'un domaine à l'autre, mais un seul est actif à un moment donné. Voici quelques exemples de domaines :

base : domaine contenant les applications minimales livrées avec *OpenMaster* ;
network : domaine contenant les applications utiles pour l'administration de réseaux ;
applications : domaine contenant les applications *AGOV* d'administration ;
help-desk : domaine contenant les applications de gestion d'un *help-desk* ;
deployement : domaine contenant les programmes nécessaires au déploiement d'applications dans un réseau ;
security : domaine contenant les applications de gestion de la sécurité d'un parc informatique ;
etc.

Le *WebTop* distingue deux types d'applications : *normales* et *génériques*. Les applications dites *normales* possèdent une icône dans la partie gauche du *WebTop* et sont exécutées lorsque l'utilisateur double-clique sur celle-ci. Les applications dites *génériques*, quant à elles, sont exécutées automatiquement lorsque l'utilisateur sélectionne le domaine auquel elles sont attachées. A tout moment, un seul domaine peut être sélectionné. Un domaine ne peut contenir qu'une seule application *générique* mais plusieurs applications *normales*. De plus, lorsqu'une application *générique* est lancée, le centre de l'applet est redécoupé en deux parties. La partie gauche est réservée à l'application *générique* et la partie droite aux applications *normales* pouvant être lancées. Bien que dans un même domaine plusieurs applications *normales* puissent être exécutées, il n'est possible d'en afficher qu'une seule à la fois. Les fenêtres des différentes applications se superposent. Seule la plus *en avant* est donc visible. Un menu, situé dans la petite boule en haut à gauche de l'applet, permet de sélectionner l'application à afficher.

Toutes les applications s'exécutant dans le *WebTop* sont des *beans* Java. Afin de décrire un bean, nous utiliserons la définition d'un composant donnée dans [Eng97, page 1]. Les

composants sont des éléments logiciels pouvant être contrôlés dynamiquement et assemblés pour former des applications. Ces composants doivent également interagir en respectant un ensemble de règles. Le modèle de composants de Java s'appelle *JavaBeans*. Un *bean* est un composant logiciel appartenant à ce modèle.

1.1.2 Le projet *Application GOVernor* (*AGOV*)

Le projet *Application GOVernor* est un projet assez particulier. En effet, il n'a pas pour objet le développement d'une application ou d'un service spécifique au sein d'*OpenMaster*. Il s'agit plutôt d'un projet ayant une vue transversale de l'ensemble du produit. Il a donc des implications dans de nombreux services et applications existants. Pour réaliser son objectif, *AGOV* nécessite un certain nombre de développements. L'un d'entre eux est la réalisation de l'application *WebTree*.

Le projet *AGOV* désire fournir aux administrateurs de systèmes les moyens de contrôler des applications du marché ou utilisateur ainsi que des systèmes d'exploitation, des bases de données, etc. Au travers d'une interface ergonomique basée sur l'architecture ouverte des services d'*OpenMaster*, *AGOV* offre donc une gestion avancée des composants critiques des systèmes d'information.

Pour ce faire, le projet *AGOV* utilise les différents services fournis par *OpenMaster* et y ajoute des fonctionnalités spécifiques. Nous retrouvons parmi celles-ci :

- une vue orientée application des différents services *OpenMaster* ;
- une ergonomie orientée vers le client et son secteur d'activité ;
- des *packages* spécifiques pour les logiciels classiques du marché (bases de données, *ERP*, Middleware[9], etc.)

AGOV a pour but d'organiser les différentes fonctionnalités d'*OpenMaster* en fonction du métier du client. Les applications existant dans *OpenMaster* sont, dans le cadre d'*AGOV*, présentées différemment suivant le contexte d'utilisation. Nous entendons par là que l'interface présentée à l'utilisateur est uniformisée et modifiée en fonction de celui-ci et du travail qu'il réalise. Ces différentes interfaces utilisent, dans la mesure du possible, des couleurs et des polices semblables.

L'originalité du produit est de proposer, au sein d'*OpenMaster*, une structure unique avec une ergonomie uniforme pour tous les composants d'un *projet applicatif du client*. On entend, par *projet applicatif*, des contextes d'utilisation tels que *réseau*, *matériel*, *logiciel*, etc. *AGOV* introduit un certain nombre de domaines applicatifs :

[9] Middleware : selon [OHE98], ensemble des logiciels nécessaires à l'interaction entre un client et un serveur.

bases de données : domaine regroupant les différents logiciels de bases de données ainsi que les bases des clients ;

systèmes d'exploitation : domaine contenant les différents systèmes d'exploitation et leurs objets administrés ;

TP : domaine regroupant les applications ayant trait aux systèmes transactionnels ;

ERP : domaine contenant des applications telles que *SAP*, etc. ;

etc.

Les caractéristiques principales d'*AGOV* sont :

- une interface *WWW* ;
- une découverte des éléments (ordinateurs, systèmes d'exploitation, *SAP*, bases de données, etc.) appartenant aux domaines d'*AGOV* ;
- une représentation des domaines sous forme d'arborescence ;
- un environnement unique pour tous les objets *AGOV* : bases de données (Oracle, Informix, Sybase), *TP*[10](Tuxedo[11]), *ERP* (*SAP*[12], Oracle Applications), JobScheduler[13], systèmes d'exploitation ;
- une ergonomie uniforme : l'interface reste cohérente et simple à utiliser à travers les différents domaines ;
- une carte des applications : représentation et animation des composants critiques ;
- un appel contextuel aux différentes applications d'*OpenMaster* ;
- des outils spécifiques exécutables au sein de leur contexte.

On peut remarquer que l'ergonomie logicielle de l'ensemble *OpenMaster* est au centre du projet *AGOV*. Il existe bien entendu quelques ajouts purement fonctionnels, mais beaucoup sont présents afin d'améliorer l'ergonomie d'utilisation offerte au client.

1.2 Les besoins

Le produit *OpenMaster* et le projet *AGOV* présentés, nous pouvons maintenant aborder la partie consacrée à la définition des besoins. Nous commencerons par définir les besoins spécifiques du projet *AGOV*. Comme nous l'avons vu dans la partie 1.1.2, ce projet utilise les différents services d'*OpenMaster*. Bien souvent, pour réaliser un objectif du projet *AGOV*, des développements sont nécessaires dans plusieurs services. Nous ne décrirons pas tous les développements demandés par *AGOV*. Dans la mesure où l'énumération et la description de l'ensemble des desiderata d'*AGOV* ne sont pas nécessaires à la bonne

[10]TP : *Transaction Processing*.
[11]TUXEDO : application transactionnelle.
[12]SAP : logiciel intégré de gestion financière, des ressources humaines et du marketing.
[13]JobScheduler : application de planification d'exécution de travaux pour Oracle.

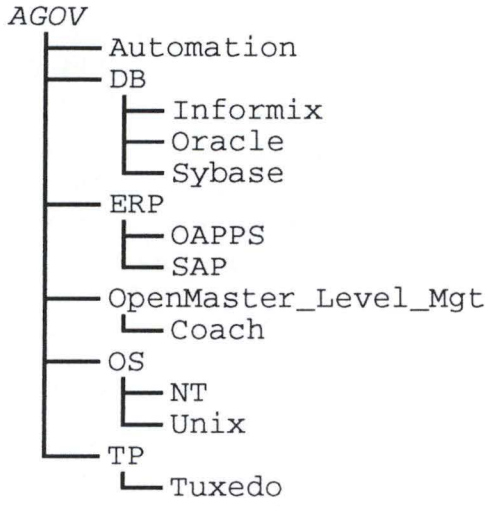


FIG. 1.3 – Exemple de domaines *AGOV*

compréhension de ce mémoire, nous nous limiterons à expliciter les besoins d'*AGOV* dans le cadre du *WebTree*. Néanmoins, certains besoins concernant les autres projets en développement affecteront, dans certains cas, les besoins et les développements du *WebTree*. Nous préciserons ces différentes implications lorsque nous aborderons la partie consacrée au développement (cf. chapitre 4, page 69).

Lorsque les besoins spécifiques au projet *AGOV* auront été décrits, nous présenterons ceux émis par d'autres projets en cours de développement. Il est en effet apparu que, dans certains cas, l'application *WebTree* pouvait répondre à des besoins provenant d'autres projets.

1.2.1 Au sein du projet *AGOV*

Parmi les caractéristiques principales d'*AGOV* que nous avons citées dans la partie 1.1.2 (page 16), nous pouvons en souligner quelques-unes :

- une interface *WWW* ;
- une représentation des domaines sous forme d'arborescence ;
- un appel contextuel à différentes applications d'*OpenMaster*.

A partir de ces caractéristiques, nous allons maintenant préciser, dans le cadre du *WebTree*, les différents besoins d'*AGOV*.

1.2.1.1 *Bean*, objets, types d'objets et noeuds

Le premier besoin exprimé par *AGOV* est d'avoir une application qui puisse être exécutée dans l'applet Java *WebTop*. Il doit donc s'agir d'un *bean* Java (cf. partie 1.1.1,

page 14). Cette application doit représenter, sous la forme d'arborescence, les différents domaines gérés par *AGOV*. Du point de vue logique, à l'intérieur de ces domaines, nous retrouvons différents objets appartenant aux domaines ; ces objets peuvent, suivant ce qu'ils représentent, contenir eux-mêmes d'autres objets. D'un point de vue graphique, l'arbre est composé de *noeuds* ; chaque noeud peut posséder des *sous-noeuds*. Un *noeud* représente donc graphiquement un objet logique.

Prenons un exemple afin d'expliquer ceci. Parmi les différents domaines d'*AGOV* que nous voyons à la figure 1.3, nous avons le domaine *DB*. Ce domaine comprend les différentes bases de données à administrer. Le domaine *DB* peut lui-même être subdivisé suivant le type de base de données. Ainsi, d'un point de vue logique, nous avons trois objets représentant les types *Informix*, *Oracle* et *Sybase*. Enfin, à l'intérieur de chacune de ces subdivisions, nous avons des objets représentant les différentes bases de données Informix, Oracle ou Sybase. D'un point de vue graphique, le *noeud* "AGOV" possède un *sous-noeud* "DB". Ce dernier a lui-même trois *sous-noeuds* : "Informix", "Oracle" et "Sybase". Enfin, sous chacun de ces noeuds, nous aurons autant de *noeuds* qu'il existera de bases de données de ces types.

De nombreux types d'objets différents doivent pouvoir être représentés. Nous aurons notamment des noeuds représentant des bases de données, des réseaux, des systèmes d'exploitation, des applications, etc. Il est aussi important de pouvoir aisément ajouter de nouveaux types de noeuds. En effet, il suffit de voir l'évolution des standards informatiques actuels, qu'il s'agisse d'applications, de bases de données ou autres, pour se rendre compte que l'on risque d'être fréquemment amené à ajouter de nouveaux domaines ou de nouveaux types de noeuds.

Il est intéressant d'avoir une hiérarchie de types de noeuds. En effet, quand on regarde les types possibles, on voit directement apparaître le fait que certains ne sont qu'une spécialisation d'autres. Ainsi, le type *base de données Oracle* est une spécialisation du type *base de données*. Certaines caractéristiques (cf. ci-dessous) sont communes à toutes les bases de données, qu'il s'agisse de bases de données Oracle, Sybase ou autres. On pourraient donc imaginer de stocker ces caractéristiques dans le type *bases de données* et de faire hériter les autres types de celui-ci. Ils hériteraient ainsi des caractéristiques et des menus de leur type père.

1.2.1.2 Caractéristiques de noeuds

Chacun des objets représentés possède des caractéristiques. Certaines caractéristiques sont évidentes :

- le nom de l'objet ;
- son icône ;
- son type ;

- etc.

D'autres sont plus particulières et leur existence dépend du noeud représenté :

- adresse réseau de l'ordinateur représenté ;
- identifiants de l'objet pour certaines applications ;
- message contextuel à afficher ;
- etc.

La liste de ces caractéristiques n'est pas exhaustive afin de faciliter l'apparition future de nouveaux objets et de nouveaux types d'objets.

Certaines caractéristiques doivent être communes à tous les objets d'un même type. Par exemple, tous les noeuds représentant des objets de type *base de données* seront représentés par la même icône. Le fait de définir l'icône dans le type de l'objet permet de ne le faire qu'une seule fois pour toutes les bases de données. Néanmoins, il doit être possible de surcharger une caractéristique dans l'objet. Ainsi, si pour une base de données spécifique on désire avoir une icône particulière, il suffit de donner une caractéristique icône à l'objet. Cette caractéristique sera préférée à celle stockée dans le type associé à cet objet.

1.2.1.3 Les actions

Les actions doivent principalement permettre de réaliser deux objectifs d'*AGOV*. Le premier est le lancement d'applications d'*OpenMaster*. Le deuxième est la découverte des éléments d'un domaine. D'autres actions doivent pouvoir être envisagées et ajoutées, éventuellement par le client. Une action doit être identifiée par un nom unique. Elle doit également contenir, pour chaque paramètre nécessaire à sa bonne exécution, le nom de ce paramètre ainsi qu'une brève description de ce qu'il représente. Cette description doit être affichée lors d'une demande de valeur à l'utilisateur.

Lors du lancement d'une application d'*OpenMaster*, un sous-noeud doit être créé en dessous du noeud sélectionné. Le menu de ce noeud doit permettre de rendre l'application associée visible (celle-ci pouvant être cachée par une autre application) ou de la fermer. Lors de la fermeture de l'application (depuis le menu contextuel ou depuis l'application), le noeud qui y est associé doit être supprimé.

Les actions de découverte (ou expansion) sont utilisées afin d'insérer dynamiquement dans l'arbre les différents éléments d'un domaine. Ainsi, on pourrait exécuter une action d'expansion sur le noeud "Oracle" afin de découvrir les différentes bases de données Oracle installées dans le parc informatique administré. L'exécution de ces actions doit correspondre à l'instanciation d'une classe Java et à l'appel d'une méthode de cette classe. Cette méthode retourne, dans un format à définir lors de l'analyse du *WebTree*, un ensemble de caractéristiques permettant à celui-ci de créer de nouveaux noeuds en dessous de celui

sélectionné. Les classes d'expansion peuvent ainsi être spécifiques à la découverte d'ordinateurs, de bases de données, de réseaux, etc. Généralement, les classes d'expansion effectuent des requêtes auprès de différents services *OpenMaster* afin d'obtenir les informations nécessaires à la création de noeuds.

Il doit également exister un certain nombre d'actions internes au *WebTree*. Ces actions doivent permettre de rendre visible ou de fermer un *bean* lancé depuis l'arbre, d'ouvrir une fenêtre du navigateur avec une *URL*[14] passée en paramètre (une des actions possibles sur un noeud étant l'affichage d'un document d'aide écrit en *HTML*[15]) et de modifier la valeur d'une caractéristique d'un noeud.

A long terme, de nouvelles actions sont destinées à voir le jour. On imagine, par exemple, des actions demandant à un service *OpenMaster* l'exécution de scripts d'installation de logiciels sur un ordinateur donné, la création d'utilisateurs sur tous les ordinateurs d'un réseau, etc. L'ajout d'une nouvelle action doit donc être relativement aisé.

1.2.1.4 Menu associé à un type de noeuds

Un autre besoin concerne la possibilité de pouvoir lancer des applications sur certains objets. Pour ce faire, lorsque l'utilisateur clique avec le bouton droit de la souris sur un noeud de l'arbre, un menu contextuel doit apparaître. Un élément du menu peut être soit une action que l'on peut exécuter sur l'objet représenté par le noeud, soit un sous-menu. Le menu contextuel doit être commun à tous les objets d'un même type. Il n'existe pas, pour *AGOV*, d'actions spécifiques à un objet en particulier. Le menu doit donc être considéré comme une caractéristique d'un type de noeuds. A un élément de menu correspondent une action et les paramètres à passer à cette action. Chaque paramètre est identifié par son nom. Ainsi, l'ordre dans lequel les paramètres sont stockés dans le menu n'a aucune importance. Le *WebTree* doit ordonner les paramètres selon les noms stockés dans la définition des paramètres de l'action.

Comme nous venons de le dire, tous les noeuds d'un même type ont le même menu. Or, lors du lancement d'une application, il est généralement intéressant de préciser l'objet sur lequel l'application doit s'exécuter. Afin de pouvoir réaliser ceci, le *WebTree* devra pouvoir procéder à une évaluation des paramètres de l'action. Il existe différents types de paramètres, chacun devant être évalué différemment :

valeur : type permettant d'affecter à un paramètre une valeur fixe pour tous les appels ;

variable : type permettant de récupérer la valeur d'une caractéristique du noeud sélectionné. La caractéristique est identifiée par son nom ;

[14] URL : *Uniform Resource Locator.*
[15] HTML : *HyperText Markup Language.*

L'édition d'une vue comporte différents volets. Afin de décrire les différentes possibilités que l'édition doit offrir, nous allons parcourir les différents éléments du *WebTree*. Pour chacun de ces éléments, nous décrirons ce qu'il doit être possible de faire.

Le *WebTree*

Au niveau du *WebTree*, il doit être possible de créer, supprimer et éditer des vues.

Les vues

Il faut pouvoir créer, supprimer, modifier, déplacer et dupliquer aisément des noeuds. Le déplacement et la duplication d'un noeud doivent pouvoir se faire à l'intérieur d'une vue, mais également entre deux vues. Il faut à ce niveau réfléchir à un moyen de limiter les possibilités de déplacements, afin de conserver une certaine cohérence. Il ne doit par exemple pas être possible de déplacer un noeud représentant l'ensemble des bases de données Oracle depuis le domaine "DB" vers le domaine "OS" censé contenir les différents systèmes d'exploitation.

Les actions

Il doit également être possible de créer, modifier et supprimer des actions ainsi que leurs paramètres et définitions.

Les noeuds

Il faut pouvoir créer, supprimer et éditer chacune des caractéristiques d'un noeud. Chaque noeud étant d'un certain type, il doit être possible d'en modifier le type.

Les caractéristiques

L'édition d'une caractéristique consiste simplement à lui affecter une valeur.

Les types de noeuds

Les différents types de noeuds doivent aussi être éditables. Cela signifie pouvoir non seulement créer, supprimer ou éditer leurs caractéristiques mais également pouvoir éditer leur menu associé.

Les menus

L'édition d'un menu doit permettre l'ajout, la suppression et l'édition de ses items et de ses éventuels sous-menus. L'édition d'un item de menu consiste à définir l'action à exécuter et ses différents paramètres.

sélectionné. Les classes d'expansion peuvent ainsi être spécifiques à la découverte d'ordinateurs, de bases de données, de réseaux, etc. Généralement, les classes d'expansion effectuent des requêtes auprès de différents services *OpenMaster* afin d'obtenir les informations nécessaires à la création de noeuds.

Il doit également exister un certain nombre d'actions internes au *WebTree*. Ces actions doivent permettre de rendre visible ou de fermer un *bean* lancé depuis l'arbre, d'ouvrir une fenêtre du navigateur avec une *URL*[14] passée en paramètre (une des actions possibles sur un noeud étant l'affichage d'un document d'aide écrit en *HTML*[15]) et de modifier la valeur d'une caractéristique d'un noeud.

A long terme, de nouvelles actions sont destinées à voir le jour. On imagine, par exemple, des actions demandant à un service *OpenMaster* l'exécution de scripts d'installation de logiciels sur un ordinateur donné, la création d'utilisateurs sur tous les ordinateurs d'un réseau, etc. L'ajout d'une nouvelle action doit donc être relativement aisé.

1.2.1.4 Menu associé à un type de noeuds

Un autre besoin concerne la possibilité de pouvoir lancer des applications sur certains objets. Pour ce faire, lorsque l'utilisateur clique avec le bouton droit de la souris sur un noeud de l'arbre, un menu contextuel doit apparaître. Un élément du menu peut être soit une action que l'on peut exécuter sur l'objet représenté par le noeud, soit un sous-menu. Le menu contextuel doit être commun à tous les objets d'un même type. Il n'existe pas, pour *AGOV*, d'actions spécifiques à un objet en particulier. Le menu doit donc être considéré comme une caractéristique d'un type de noeuds. A un élément de menu correspondent une action et les paramètres à passer à cette action. Chaque paramètre est identifié par son nom. Ainsi, l'ordre dans lequel les paramètres sont stockés dans le menu n'a aucune importance. Le *WebTree* doit ordonner les paramètres selon les noms stockés dans la définition des paramètres de l'action.

Comme nous venons de le dire, tous les noeuds d'un même type ont le même menu. Or, lors du lancement d'une application, il est généralement intéressant de préciser l'objet sur lequel l'application doit s'exécuter. Afin de pouvoir réaliser ceci, le *WebTree* devra pouvoir procéder à une évaluation des paramètres de l'action. Il existe différents types de paramètres, chacun devant être évalué différemment :

valeur : type permettant d'affecter à un paramètre une valeur fixe pour tous les appels ;

variable : type permettant de récupérer la valeur d'une caractéristique du noeud sélectionné. La caractéristique est identifiée par son nom ;

[14] URL : *Uniform Resource Locator.*
[15] HTML : *HyperText Markup Language.*

concaténation : type permettant de concaténer plusieurs valeurs obtenues par l'évaluation d'une suite de paramètres ;

vecteur : type permettant de définir un paramètre de type vecteur ainsi que ses différents éléments. Chacun des éléments est lui-même un paramètre évalué ;

requête : type permettant d'obtenir la valeur d'un paramètre en interrogeant un service spécifique d'*OpenMaster*. La compréhension de ce service n'étant pas nécessaire à la compréhension du mémoire, nous ne le décrirons pas.

Lorsque l'utilisateur clique sur un élément de menu, le *WebTree* doit procéder à l'évaluation des différents paramètres. Ensuite, l'action peut être exécutée. Tous les paramètres requis pour l'exécution de l'action ne doivent pas obligatoirement être spécifiés dans la configuration d'appel. Si un ou plusieurs paramètres sont omis, il faut demander leur valeur à l'utilisateur.

1.2.1.5 L'animation

OpenMaster possède un système d'alarmes notamment utilisé par l'affichage des icônes situées dans la partie droite du *WebTop* (cf. figure 1.2, page 14). Il sert également à représenter l'état de différents éléments administrés (réseaux, ordinateurs, bases de données, etc.). Lorsqu'une icône change de couleur en fonction de l'état de l'objet qu'elle représente, on dit que l'icône est animée.

AGOV désire également utiliser les possibilités d'animation dans l'arbre. L'utilisateur aura ainsi une vision plus aisée de l'ensemble de ses systèmes et de leurs états. De plus, le client n'aura plus à démarrer les différentes applications d'*OpenMaster* capables de fournir une représentation par animation puisqu'il aura, dans le *WebTree*, une vision des différents objets animés.

Il doit donc être possible de configurer, démarrer et arrêter l'animation sur un noeud. Seuls des noeuds représentant des objets existant dans le système d'alarmes peuvent être animés.

La configuration de l'animation doit comprendre, en plus de la configuration propre au système d'alarmes, la possibilité de définir, parmi les différentes caractéristiques, les icônes à utiliser pour signaler l'état d'un noeud.

1.2.1.6 Les vues

Le projet *AGOV* se veut orienté *métier*. On entend par là le fait d'offrir au client une interface et une manière de travailler avec *OpenMaster* les plus proches possible de

son métier. Ceci est d'autant plus intéressant dans le cadre d'une entreprise de grande envergure où le nombre de systèmes hétérogènes à administrer est grand. Dans ce cas, l'administration n'est bien entendu pas confiée à une seule personne. Certaines personnes sont chargées de la gestion du réseau, d'autres des bases de données, d'autres encore des applications, etc. De même, dans chacune de ces catégories, il peut exister des subdivisions, soit en fonction des connaissances de l'administrateur, soit en fonction de la division dans laquelle il se trouve. Ainsi, un administrateur de bases de données Oracle n'est intéressé ni par les autres bases de données, ni par la gestion du réseau. De plus, si cet administrateur se trouve à Bruxelles, il n'administre certainement que les bases du siège bruxellois et pas celles de Paris, San Francisco, etc.

Dans ce cadre, il est judicieux de pouvoir fournir à l'utilisateur une vue propre à son métier. Une vue est un arbre à part entière représentant les noeuds qui intéressent l'utilisateur dans la réalisation de son travail. Ainsi, on imagine très bien une vue *bases de données*, une vue *réseaux*, une vue *ERP*, etc. Il peut également être intéressant pour l'utilisateur d'avoir une vue représentant uniquement l'application dont il a la charge.

Au démarrage de l'application *WebTree*, toutes les vues ne doivent pas être chargées. Lors du premier démarrage de l'application, la vue d'ensemble est présentée. L'utilisateur peut à loisir la fermer et / ou en ouvrir d'autres. Aux lancements suivants, le *WebTree* doit recharger automatiquement les vues ouvertes lorsque l'utilisateur a quitté le *WebTop*. L'utilisateur doit bien entendu pouvoir passer d'une vue à l'autre très simplement. Chaque vue est identifiée par un nom. Il peut être intéressant de pouvoir afficher simultanément deux vues distinctes.

1.2.1.7 L'édition des vues

Comme nous venons de le voir, le *WebTree* doit offrir la possibilité d'utiliser des vues. Le projet *AGOV* peut présenter aux clients un certain nombre de vues prédéfinies. Celles-ci seront stockées dans des fichiers texte selon un format à déterminer lors du développement de l'application. Il n'est cependant ni possible ni réaliste de concevoir toutes les vues dont les clients pourraient avoir besoin. Celles-ci dépendront en effet non seulement des systèmes que le client désirera administrer, mais également de l'organisation interne de l'entreprise.

Il est donc très important de pouvoir offrir aux clients la possibilité de créer eux-mêmes des vues correspondant tant à leur parc informatique qu'à leur organisation. C'est pourquoi le *WebTree* doit permettre l'édition des vues.

Cette capacité d'édition est également intéressante au sein même des équipes de développement d'*OpenMaster*. En effet, les différentes vues prédéfinies toucheront à des domaines d'administration différents. Ces vues ne seront pas toutes réalisées par le projet AGOV. Certaines, propres à un domaine *AGOV*, seront créées par une équipe travaillant dans ce domaine.

L'édition d'une vue comporte différents volets. Afin de décrire les différentes possibilités que l'édition doit offrir, nous allons parcourir les différents éléments du *WebTree*. Pour chacun de ces éléments, nous décrirons ce qu'il doit être possible de faire.

Le *WebTree*

Au niveau du *WebTree*, il doit être possible de créer, supprimer et éditer des vues.

Les vues

Il faut pouvoir créer, supprimer, modifier, déplacer et dupliquer aisément des noeuds. Le déplacement et la duplication d'un noeud doivent pouvoir se faire à l'intérieur d'une vue, mais également entre deux vues. Il faut à ce niveau réfléchir à un moyen de limiter les possibilités de déplacements, afin de conserver une certaine cohérence. Il ne doit par exemple pas être possible de déplacer un noeud représentant l'ensemble des bases de données Oracle depuis le domaine "DB" vers le domaine "OS" censé contenir les différents systèmes d'exploitation.

Les actions

Il doit également être possible de créer, modifier et supprimer des actions ainsi que leurs paramètres et définitions.

Les noeuds

Il faut pouvoir créer, supprimer et éditer chacune des caractéristiques d'un noeud. Chaque noeud étant d'un certain type, il doit être possible d'en modifier le type.

Les caractéristiques

L'édition d'une caractéristique consiste simplement à lui affecter une valeur.

Les types de noeuds

Les différents types de noeuds doivent aussi être éditables. Cela signifie pouvoir non seulement créer, supprimer ou éditer leurs caractéristiques mais également pouvoir éditer leur menu associé.

Les menus

L'édition d'un menu doit permettre l'ajout, la suppression et l'édition de ses items et de ses éventuels sous-menus. L'édition d'un item de menu consiste à définir l'action à exécuter et ses différents paramètres.

Toujours dans un souci d'ergonomie et de facilité d'utilisation, l'édition d'une vue doit être la plus simple possible. A cette fin, il peut être intéressant d'utiliser des techniques telles que le *Drag&Drop*, pour autant que cela soit possible avec Java.

Une remarque importante est que le *WebTree* ne doit pas veiller, durant l'édition, à l'intégrité des différentes vues. L'utilisateur est considéré comme compétent. Ainsi, si un type de noeuds est supprimé alors qu'il est encore utilisé dans certaines vues, celles-ci deviennent inutilisables et le *WebTree* ne doit donc plus pouvoir les charger.

1.2.1.8 L'internationalisation

OpenMaster est une application multilingue. Le *WebTree* doit aussi l'être. Ainsi, tous les affichages, qu'il s'agisse des menus ou du nom de chaque noeud, doivent être dépendants de la langue et du pays où le navigateur est exécuté. Ceci est vrai tant pour le français, l'anglais, etc. que le japonais. Bien entendu, on ne demande pas de traduire les affichages. Il doit être possible de fournir des fichiers de ressources contenant les traductions pour les différentes langues.

1.2.1.9 Une *API* [16]

Dans le cadre de projets regroupant plusieurs services, certaines applications pourraient désirer créer elles-mêmes des noeuds dans l'arbre. Elles pourraient également vouloir modifier des caractéristiques de certains noeuds. De même, certains clients peuvent, grâce aux différentes *API*'s d'*OpenMaster* mises à leur disposition, développer différents modules spécifiques à leurs besoins. La différence majeure avec les besoins d'édition décrits ci-dessus est que dans l'édition, l'acteur principal est l'utilisateur d'*OpenMaster* alors qu'ici, l'utilisateur ne réalise aucune action particulière. Il utilise seulement les différentes applications d'*OpenMaster* qui elles-mêmes agissent sur le *WebTree*.

Afin de répondre à cela, le *WebTree* doit fournir un ensemble d'*API*'s Java afin de permettre à différentes applications d'utiliser l'arbre pour leurs propres besoins.

Les différentes fonctionnalités requises sont :

- la création et la suppression d'un noeud dans une vue ;
- la consultation et la modification d'une caractéristique d'un noeud ;
- la modification du type d'un noeud ;
- la création de nouveaux types de noeuds ;
- la consultation et la modification d'un type de noeuds ;
- la création de nouvelles actions.

[16] API : *Application Programming Interface.*

1.2.1.10 Une application générique

Finalement, l'application *WebTree* doit être une application *générique*. Ainsi, dès que l'utilisateur entre dans le domaine *OpenMaster* d'*AGOV*, l'application est affichée et l'utilisateur peut commencer à travailler.

1.2.2 Dans les autres projets

Les desiderata propres au commanditaire de l'application *WebTree* étant décrits, nous allons maintenant aborder ceux provenant des autres projets existant au sein d'*OpenMaster*. Bien que dans certains cas il s'agisse de besoins spécifiques à un projet, il sera généralement question de besoins généraux pour l'ensemble des applications d'*OpenMaster*.

Bien entendu, tout comme dans la partie précédente, nous ne présenterons que les besoins auxquels le *WebTree* pourrait répondre. Encore une fois, il n'est nullement question de décrire ici tous les développements et projets en cours dans *OpenMaster*.

1.2.2.1 Uniformisation des fichiers de configuration

Un premier besoin général à *OpenMaster* est la nécessité d'uniformiser les différents formats des fichiers de configuration des applications, pour lesquels il n'existe aucune convention. Ceci a pour conséquence que chaque programme définit son format en fonction de ses besoins. Néanmoins, les besoins de chacun étant souvent fort différents, il serait difficile de définir un format unique.

Prenons un exemple simple. Comme nous l'avons vu dans les besoins d'*AGOV*, le *WebTree* doit modéliser des actions. Ainsi, dans ses fichiers de configuration, nous retrouverons des descriptions d'actions. Une autre application d'*OpenMaster - Configuration Manager* - utilise également des actions. Les fichiers de configuration de cette application contiennent également des descriptions d'actions. Or, les deux applications sont différentes. On peut donc aisément comprendre que la définition d'une action dans le cadre du *WebTree* soit différente de la définition d'une action dans le cadre de l'autre application. Les données enregistrées dans les différents fichiers de configuration des deux applications sont donc également différentes.

Le but recherché dans l'uniformisation des fichiers de configuration n'est pas d'obliger les différentes applications à modéliser les différents objets de la même façon mais bien de leur fournir une manière unique d'enregistrer leur configuration, quel que soit le modèle de données qu'elles utilisent. Comme nous venons de le voir, il est normal que ces modélisations soient différentes. On ne veut pas uniformiser la modélisation des données, mais le format de représentation de cette modélisation dans les fichiers.

Il existe de multiples intérêts à uniformiser les fichiers de configuration. Un des intérêts principaux est le fait d'avoir une compréhension de ces fichiers plus aisée par tout le monde. De plus, il serait alors possible de créer des outils afin d'éditer facilement ces différents fichiers. Pour le moment, il faudrait un programme d'édition par fichier de configuration.

Le *WebTree* étant une nouvelle application, il est nécessaire de lui définir un format de fichier. Il serait donc intéressant de profiter de cette opportunité pour définir un modèle de représentation dans les fichiers de configuration.

1.2.2.2 Gestion des rôles

Afin d'augmenter les possibilités de sécurité d'*OpenMaster*, il a été décidé d'introduire une nouvelle notion dans *OpenMaster* : la notion de *rôle*. Nous ne décrirons pas dans tous ses détails cette nouvelle fonctionnalité d'*OpenMaster*, nous nous limiterons à décrire le strict nécessaire à la compréhension des besoins dans ce domaine. Cette notion a pour but d'attribuer un rôle à chaque utilisateur d'*OpenMaster*. Un rôle est donc une liste de privilèges attribués à un utilisateur. A chaque action d'une des applications *OpenMaster* doit être associé un privilège d'exécution pour cette action.

Lorsqu'un utilisateur démarre le *WebTop*, une fenêtre de *login* lui est présentée. Cette fenêtre demande un mot de passe (le nom de l'utilisateur est récupéré au niveau du système d'exploitation) et le rôle que l'utilisateur désire jouer. Une fois l'accès de l'utilisateur validé, celui-ci peut commencer à travailler. Le fait d'être connecté avec tel ou tel rôle lui octroie l'accès (en lecture, écriture, suppression, etc.) aux différentes applications et à leurs objets. Ainsi, si un utilisateur se connecte avec le rôle *administrateur de bases de données*, il ne peut utiliser que les programmes ayant trait à l'administration de bases de données ainsi que les objets représentant des bases de données. Il ne peut par contre pas accéder aux programmes d'administration de systèmes ou de réseaux.

Bien entendu, l'utilisateur ne peut pas choisir n'importe quel rôle. Il ne peut sélectionner qu'un des rôles que l'administrateur d'*OpenMaster* lui a attribués. Il est tout à fait possible qu'un utilisateur n'ait qu'un seul rôle. Il doit donc exister une application permettant à l'administrateur d'*OpenMaster* de configurer l'ensemble des rôles et des droits d'accès aux différentes ressources. Cette application doit également posséder un certain nombre de fichiers de configuration.

Les besoins émis par la gestion des rôles se situent à deux niveaux. L'un est plus centré sur la gestion des rôles au sein du *WebTree*, l'autre se situe plus au niveau de la modélisation des rôles. Voyons ces deux besoins un peu plus en détail.

Le premier se situe donc au sein du *WebTree*. La notion de rôle a un certain nombre d'implications dans les différentes applications. En effet, il est nécessaire de modifier les applications afin qu'elles intègrent ce nouveau concept. En ce qui concerne le *WebTree*, cela

signifie qu'il faut intégrer la notion de rôle à tous les objets représentés ainsi qu'au niveau des actions exécutables sur ces objets. En effet, un administrateur de bases de données ne devra plus avoir accès aux vues d'administration de réseaux, etc. De même, bien qu'il puisse toujours voir les ordinateurs où sont installées les bases de données, il ne doit pas pouvoir exécuter des actions d'administration de ces machines si ces actions n'ont aucun lien avec son métier premier, la gestion des bases de données. Lorsque l'utilisateur désire exécuter une action, le *WebTree* doit vérifier auprès du gestionnaire de rôles si l'utilisateur possède le privilège nécessaire pour pouvoir exécuter l'action.

Le deuxième besoin concerne plutôt une utilisation commune des développements et recherches menés dans le cadre du *WebTree*. Il s'agit ici de voir s'il ne serait pas possible d'utiliser le même modèle que celui qui sera choisi pour le *WebTree* afin de modéliser les différents objets intervenant dans la gestion des rôles et donc d'avoir un format de sauvegarde qui pourrait être commun avec celui de l'arbre. De plus, il pourrait être intéressant, si les fonctionnalités du *WebTree* sont suffisamment génériques, de pouvoir éditer les fichiers de configuration au sein d'une vue du *WebTree*.

1.2.2.3 Les actions conditionnelles

Dans la description des besoins que nous avons faite, chaque type de noeuds possède un menu décrivant les actions pouvant être exécutées sur les noeuds de ce type. Ceci est donc réalisé lors de la création ou de la modification du type de noeuds. Or, certaines applications aimeraient pouvoir dynamiquement dire qu'elles peuvent être appelées sur tel ou tel noeud (et pas type de noeuds). Il faudrait donc, lorsqu'un noeud est sélectionné, que le *WebTree* demande à ces applications si elles ont une action à faire apparaître dans le menu contextuel du noeud.

1.2.2.4 Configuration Manager

L'application *Configuration Manager* est utilisée dans le cadre du déploiement et de la configuration de logiciels. Pour réaliser cette tâche, elle utilise le concept d'actions qui sont exécutées sur un ou plusieurs éléments du parc informatique. Il existe, par exemple, une action de création d'utilisateur. Cette action peut être appliquée sur des ordinateurs afin de créer un utilisateur donné sur chacun d'eux.

Les différents ordinateurs pouvant être représentés dans le *WebTree*, il serait intéressant de pouvoir lancer des actions de *Configuration Manager* depuis celui-ci. Il faudrait donc pouvoir créer des actions *Configuration Manager* dans l'arbre et pouvoir les appliquer sur ses différents éléments.

1.3 Conclusion

Nous voici arrivé au terme de ce premier chapitre durant lequel nous avons pu nous familiariser avec le cadre d'existence de ce mémoire. Nous avons ainsi décrit le produit *OpenMaster*, et plus particulièrement le *WebTop* et le projet *AGOV*. Nous avons également présenté les différents besoins du projet *AGOV*, besoins qui sont à la base même du développement du *WebTree*. Enfin, nous avons vu les desiderata des autres projets, desiderata qui influenceront notre travail.

Ces différents concepts étant définis, nous pouvons présenter l'application *WebTree* et décrire comment elle répond aux différents besoins. Ce sera l'objet du chapitre suivant.

Chapitre 2

L´application *WebTree*

Après avoir décrit dans le chapitre précédent l'ensemble des besoins ayant conduit à la réalisation de l'application *WebTree*, nous allons maintenant nous attacher à la décrire le plus clairement et complètement possible. Nous montrerons ainsi comment l'application que nous avons réalisée répond aux différents besoins. Nous ne nous engagerons néanmoins pas dans une description technique de l'application. Ainsi, nous ne décrirons pas comment telle ou telle partie du *WebTree* est implémentée mais plutôt les différents services offerts à l'utilisateur de notre application.

Nous commencerons par expliciter les différents concepts intervenant dans le *WebTree*. Ceux-ci découlent de l'analyse que nous avons effectuée pour la réalisation de l'application. Ils sont donc la base de tout le développement de l'application. Leur description permettra de mieux comprendre le point suivant de ce chapitre, qui sera la description de l'interface utilisateur du *WebTree*. Ne pas les décrire serait équivalent à présenter l'interface d'un tableur sans expliquer ce que sont les notions de cellule, de formule, etc.

Dans la partie suivante consacrée à la description de l'interface utilisateur, nous montrerons comment tous les concepts du *WebTree* sont représentés et comment l'utilisateur peut aisément travailler sur base de ceux-ci.

Avant de pouvoir entamer la description du *WebTree*, il nous reste une remarque à exprimer. Parmi les besoins décrits précédemment, certains, comme l'*API* ou la nécessité d'avoir un format de fichier unique, sont d'ordre plutôt techniques. Or, comme nous venons de le dire, la description que nous allons réaliser n'abordera pas la partie technique du *WebTree*. Il est donc clair que ce chapitre ne décrira nullement comment le *WebTree* répondra à ces besoins. Ceci sera abordé dans les chapitres suivants.

2.1 Les concepts du *WebTree*

La première partie de ce chapitre va donc être consacrée à présenter les différents concepts propres au fonctionnement du *WebTree*. Ces différents concepts sont apparus

durant l'analyse que nous avons réalisée lors de notre stage. L'analyse des besoins et de la façon dont nous allions y répondre ayant occupé une grande partie de notre stage, il serait trop long de décrire complètement le processus qui nous a mené à ces concepts. Nous nous limiterons à utiliser le résultat de ce processus pour décrire l'application *WebTree* et ses concepts.

Bien qu'ayant joui d'une grande liberté quant au déroulement de l'analyse, nous rendions régulièrement compte de son avancement et de la manière dont nous concevions le *WebTree*. Dans ce cadre, nous avons décidé d'utiliser *UML* afin d'avoir un langage de modélisation commun et donc de faciliter la compréhension par chacun du modèle que nous élaborions. C'est pourquoi nous allons également, dans le cadre de ce mémoire, utiliser des schémas *UML* afin de décrire les différents concepts du *WebTree*. Néanmoins, pour en faciliter la compréhension, et ne pas limiter celle-ci aux personnes connaissant *UML*, nous commencerons par décrire brièvement ce qu'est *UML*. Une fois cette description réalisée, nous en viendrons au sujet premier de ce chapitre : la description des concepts du *WebTree*.

2.1.1 *UML*

Unified Modeling Language (*UML*) est un langage destiné à la spécification, la visualisation, la construction et la documentation des composants logiciels aussi bien qu'à la modélisation des systèmes d'affaires ou d'autres systèmes non informatiques. Dans notre cas, nous l'avons utilisé afin de modéliser l'application *WebTree* et son fonctionnement. Nous ne décrirons complètement ici ni la sémantique d'*UML* ni sa notation. Nous nous limiterons à une description suffisante à la bonne compréhension tant des raisons qui nous ont poussé à l'utiliser que des schémas que nous présenterons plus loin dans ce chapitre. Nous nous baserons principalement sur [UML97a] et sur [UML97b] afin de réaliser cette description.

Avant la création d'*UML*, aucun des langages de modélisation orientée objets existant n'était universellement reconnu et utilisé. Les personnes désirant modéliser un système devaient tout d'abord choisir quel langage utiliser. Ce choix devait être fait entre de nombreux langages qui divergeaient généralement très peu d'un point de vue conceptuel. Le manque de cohésion entre ces divers langages décourageait souvent de nouveaux utilisateurs à entrer dans la conception orientée objets et plus particulièrement dans la modélisation orientée objets. De même, peu de créateurs de logiciels se lançaient dans la réalisation d'outils d'aide à la modélisation vu le nombre élevé de langages distincts qu'il aurait fallu supporter afin de pouvoir toucher un maximum d'utilisateurs potentiels et donc rentabiliser les développements.

2.1.1.1 Les buts d'*UML*

Rappelons brièvement les buts premiers, tels qu'explicités par les concepteurs lors de la réalisation d'*UML* :

1. fournir aux utilisateurs un langage de modélisation visuel, expressif et prêt à être utilisé de sorte qu'ils puissent développer et s'échanger des modèles significatifs ;
2. fournir des mécanismes permettant d'étendre et de spécialiser les concepts de base du langage ;
3. être indépendant de tout langage de programmation et de tout processus de développement ;
4. fournir une base formelle pour la compréhension du langage de modélisation ;
5. encourager la croissance du marché des outils de conception orientée objets ;
6. supporter des concepts de développement de plus haut niveau tels que la collaboration, les *frameworks* et les composants ;
7. intégrer les meilleures pratiques existantes.

2.1.1.2 La notation

Nous ne décrirons pas toute la sémantique des différents éléments qui composent *UML*. Nous nous limiterons à présenter la signification et la notation des quelques concepts utilisés dans les schémas que nous emploierons dans la suite de ce mémoire. Ainsi, la description de certains éléments d'*UML* restera incomplète par rapport à leur spécification. Afin de ne pas introduire d'ambiguïté dans la présentation des différents concepts *UML*, nous les nommerons en utilisant les termes anglais.

Class

L'élément *class* représente un ensemble d'objets qui partagent les mêmes *attributes*, *operations*, *methods*, *relationships* et *semantics*. Nous expliciterons plus tard le concept d'*attribute*. Une *operation* est un service qui peut être demandé à un objet afin d'en modifier le comportement. Une *method* est quant à elle l'implémentation d'une *operation*. Le concept de *relationship* sera également détaillé plus loin en abordant les concepts d'*association* et de *generalization*. Une *class* peut également utiliser un ensemble d'*interfaces* pour spécifier une collection d'*operations* qu'elle met à la disposition de son environnement. On dira alors que cette *class* "réalise" ou "implémente" l'*interface*. Le schéma a) de la figure 2.1 nous présente une *class* appelée "Vehicule" qui possède deux attributs : "marque" et "cylindree".

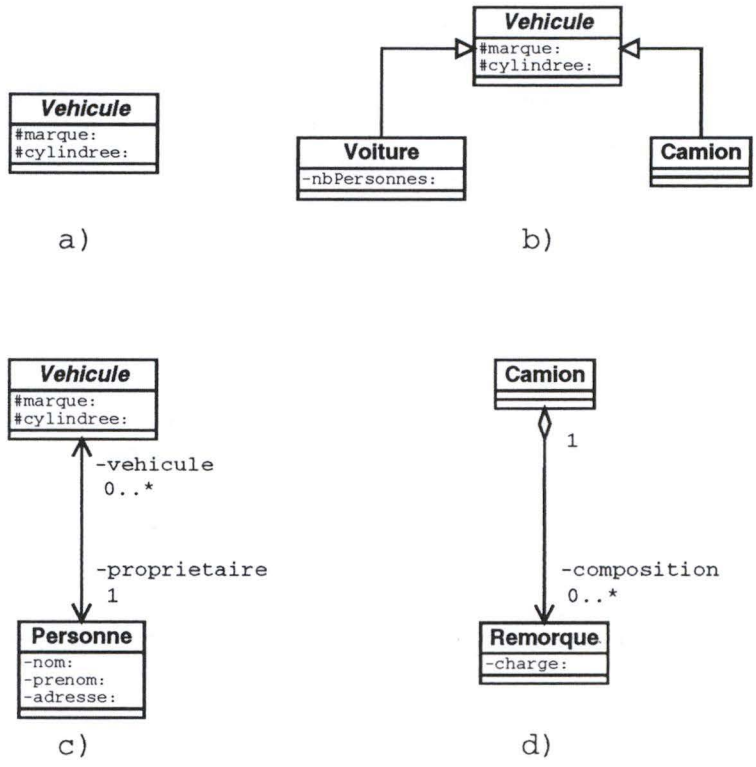


FIG. 2.1 – La notation *UML*

Association

L'élément *association* représente une relation sémantique (*semantic relationship*) entre deux *classes*. Une *association* consiste en au moins deux *AssociationEnds* qui représentent chacune une connexion entre l'*association* et une *class*. Chaque *AssociationEnd* peut posséder un ensemble de propriétés qui doivent toutes être remplies pour que l'*association* soit valide.

Les différentes propriétés d'une *AssociationEnd* que nous utiliserons sont :

1. *aggregation* : dans une même *association*, une seule *AssociationEnd* peut posséder cette propriété. Elle est représentée par un losange et signifie que la *class* de cette *AssociationEnd* est un agrégat des *classes* des autres *AssociationEnds*. Ainsi, sur le schéma d) de la figure 2.1, on voit apparaître le fait qu'un *Camion* est un agrégat d'instances *Remorque*.

2. *multiplicity* : représente le nombre d'instances (ou encore cardinalité) de la *class* de l'*AssociationEnd* qui peuvent être associées à une et une seule *class* source au travers de l'*association*. Toujours sur le schéma d), on peut voir qu'à un *Camion* sont associées entre 0 et n (représenté par le caractère '*') *Remorques* tandis qu'une *Remorque* n'est associée qu'à un seul *Camion*. Sur le schéma c) un *Vehicule* est

associé à une seule *Personne* et à une *Personne* sont associés de 0 à n *Vehicules*. Si aucune *multiplicity* n'est représentée graphiquement, on considère qu'il s'agit de 0 à n.

3. *isNavigable* : spécifie si on peut parcourir l'*association* depuis une *class* d'une *AssociationEnd* jusqu'à la *class* de l'*AssociationEnd* qui possède cette propriété. Cette propriété est représentée graphiquement par une flèche sur l'association. On voit sur le schéma c) que l'*association* est *navigable* dans les deux sens. Par contre, sur le schéma d), l'*association* ne peut être parcourue que de la classe "Camion" vers la classe "Remorque".

4. *name* : représente le nom du rôle joué par l'*AssociationEnd*. Par exemple, sur le schéma d) une *Remorque* joue le rôle de *composition* dans l'*association* entre les *classes* "Camion" et "Remorque".

Generalization

Une *generalization* représente une relation entre un élément plus général et un élément plus spécifique. Ce dernier hérite de toutes les propriétés, méthodes, etc. de l'élément plus général. On dira également que l'élément plus spécifique dérive du plus général. Sur le schéma b) les *classes* "Voiture" et "Camion" héritent de la *class* "Vehicule". De plus, la *class* "Vehicule" est une classe abstraite qui ne peut donc être instanciée. Ceci est noté sur les différents schémas par l'utilisation de l'italique dans le nom de la classe.

Attribute

Un *attribute* représente une propriété que les instances d'une *class* peuvent avoir. Chacune de ces propriétés possède un nom et une cardinalité. Celle-ci est généralement de *1..1*, auquel cas elle n'est pas mentionnée graphiquement. De plus, il est possible de limiter la visibilité d'un *attribute*. Les quatre catégories de visibilité sont :

1. *public* : l'*attribute* est accessible pour tous les objets. Dans la notation, on fait alors précéder le nom de l'*attribute* par le caractère '+' ;

2. *protected* : l'*attribute* n'est accessible qu'aux instances de la *class* ou aux instances des *classes* dérivées. Le caractère '#' est utilisé devant le nom de l'*attribute* pour représenter le fait qu'il est *protected* ;

3. *private* : l'*attribute* n'est accessible qu'aux instances de la *class*. Dans ce cas, c'est le caractère '-' qui est utilisé pour représenter la visibilité de l'*attribute* ;

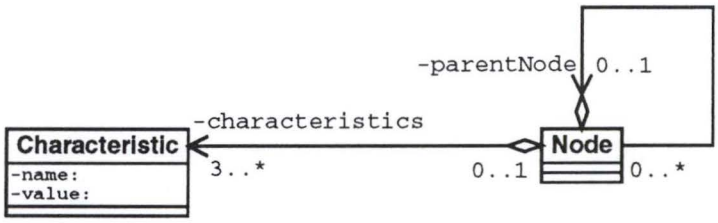


FIG. 2.2 – Le *Node*

4. *implementation* : certains langages comme Java ont une notion de *package*. Un *package* regroupe un ensemble de *classes* ayant un lien sémantique entre elles. Un *attribute* ayant une visibilité de type *implementation* n'est accessible qu'aux instances de *classes* appartenant au même *package*. Si aucun caractère ne précède le nom de l'*attribute*, alors celui-ci a une visibilité de type *implementation*.

On remarquera sur la figure 2.1 que la *class* "Vehicule" possède deux *attributes* : "marque" et "cylindree". Ceux-ci ont une visibilité de type *protected*. L'*attribute* "nbPersonnes" de la *class* "Voiture" est quant à lui de type *private*. Il en est de même pour les *attributes* des *classes* "Personne" et "Remorque".

2.1.2 Au coeur des concepts

Ces quelques notions d'*UML* décrites, nous pouvons maintenant nous tourner vers le premier objectif de ce chapitre : la présentation des différents concepts sur lesquels se base toute l'architecture du *WebTree*. Afin de réaliser cette présentation, nous partirons des concepts les plus simples sur lesquels nous nous baserons alors pour en expliquer d'autres plus complexes.

Les schémas *UML* présentés pour chaque concept ne montreront pas toutes les relations existantes mais seulement celles directement en rapport avec le concept abordé. Il est ainsi tout à fait possible qu'une relation d'agrégation entre deux classes soit montrée dans un schéma et que cette même relation n'apparaisse pas dans un autre. De même, seuls les attributs nécessaires à la présentation des concepts apparaîtront. Enfin, les opérations définies dans les différentes classes ne seront pas représentées[1].

2.1.2.1 Le *Node*

Le premier concept abordé est celui de *Node*, représenté par le schéma *UML* de la figure 2.2. Il représente un noeud de l'arbre.

[1]Le schéma complet de l'architecture du *WebTree* est fourni en annexe.

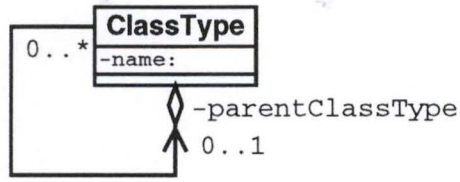


FIG. 2.3 – Le *ClassType*

Comme nous pouvons le remarquer sur le schéma, un noeud (classe *Node*) possède un certain nombre de caractéristiques (classe *Characteristic*). Chaque caractéristique possède un nom *-name-* et une valeur *-value*. Notons qu'un noeud ne peut avoir plusieurs caractéristiques de même nom. De plus, chaque noeud possède au moins trois caractéristiques : *name*, *classtype* et *nodetype*. La caractéristique *name* détermine le nom attribué à un noeud et donc le nom affiché dans l'arbre pour représenter celui-ci. Les deux autres caractéristiques seront respectivement décrites aux paragraphes 2.1.2.2 et 2.1.2.5.

Chaque noeud, à l'exception de la racine de l'arbre, possède également un père. Ceci est matérialisé dans le schéma par la relation d'agrégation dont les deux extrémités sont situées sur la classe *Node*.

Enfin, le fait qu'à une *Characteristic* soit associé 0 ou 1 noeud est dû à l'existence d'une autre association que nous verrons au point 2.1.2.5.

2.1.2.2 Le *ClassType*

Le *ClassType*, présenté à la figure 2.3, représente une catégorie de noeuds. Ce concept a pour but de subdiviser l'ensemble des noeuds affichés en diverses catégories logiques. Ces catégories seront utilisées afin de définir des règles pour réaliser le *Drag&Drop* entre deux noeuds de l'arbre. Ce concept de règle sera abordé dans la partie 2.1.2.6. Toute catégorie possède un nom unique. Ainsi, des exemples de catégorie pourraient être "machine", "réseau", "bases de données", etc.

De plus, une catégorie peut-être subdivisée en sous-catégories. Ainsi, on pourrait imaginer des sous-catégories "machine Unix", "machine NT", etc. qui auraient comme *parentClassType* la catégorie "machine". On peut considérer que l'ensemble des catégories forme une arborescence dont la racine est la catégorie "root". Cette catégorie est la seule à ne pas avoir de *parentClassType*.

2.1.2.3 Le *Parameter*

Comme nous l'avons évoqué dans la description des besoins, l'utilisateur doit pouvoir exécuter des actions sur un noeud du *WebTree*. Une action possède généralement un en-

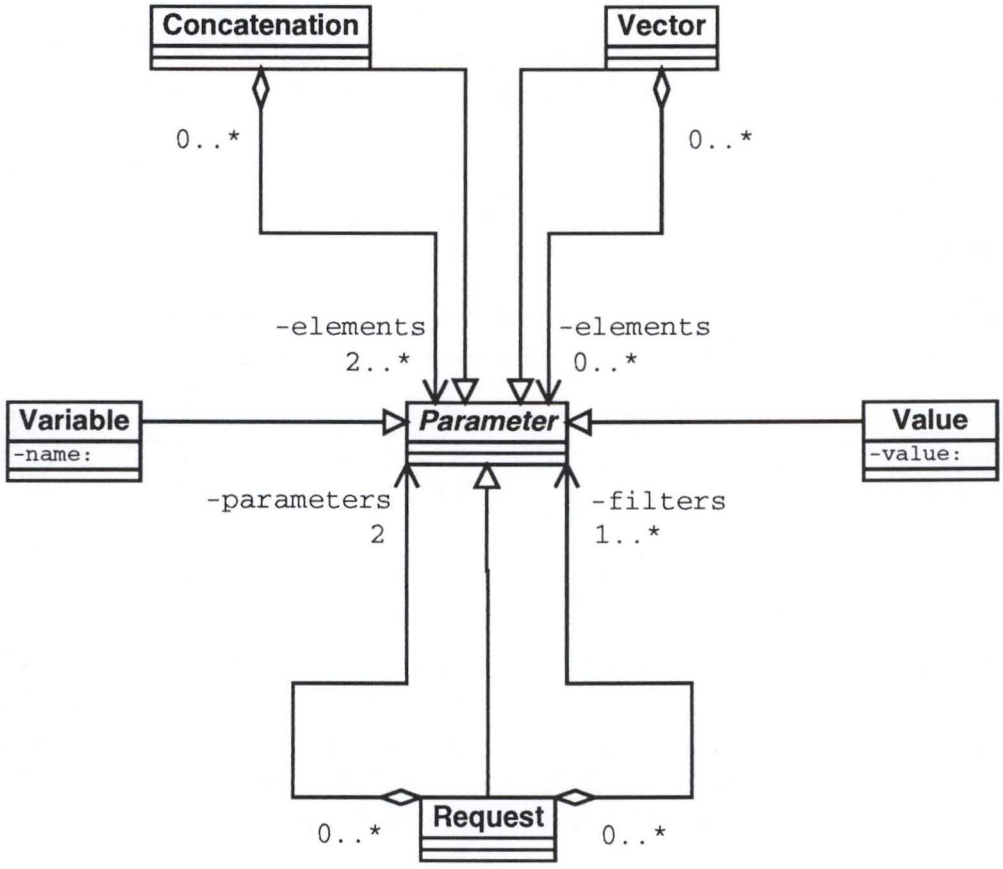


FIG. 2.4 – Le *Parameter*

semble de paramètres qui déterminent son contexte d'exécution (par exemple dans le noeud sur lequel on désire exécuter l'action). Or, un nombre important de noeuds seront créés dynamiquement durant l'utilisation du *WebTree* (cf. 1.2.1.3, page 20 : actions, mécanisme de découverte, etc.). Il n'est donc pas possible lors de la définition d'un menu de spécifier toutes les valeurs qui seront passées aux paramètres : certaines valeurs seront peut-être connues, d'autres ne le seront qu'au moment de l'exécution. Néanmoins, on sait que, par exemple, pour un paramètre donné, il faut passer le nom du noeud, pour un autre, il faut passer un vecteur contenant diverses valeurs connues ainsi que la valeur d'une des caractéristiques du noeud sélectionné. Le concept de *Parameter* représenté par le schéma de la figure 2.4 permet de spécifier comment la valeur d'un paramètre sera calculée lors de l'exécution de l'action. La classe "Parameter" ne peut jamais être instanciée. Il s'agit uniquement d'une classe abstraite dont dérivent toutes les classes pouvant être utilisées comme paramètres.

Comme nous pouvons le voir sur le schéma, il existe différentes manières de calculer la valeur d'un paramètre. Tout d'abord, la valeur peut être fixe et connue lors de la configuration du menu déroulant associé à un noeud (cf. 2.1.2.5). Nous utilisons alors la classe *Value* dont l'attribut *value* contiend la valeur à donner à un paramètre.

La valeur d'un paramètre peut également être la valeur d'une caractéristique d'un noeud. Dans ce cas, nous ferons appel au concept de *Variable*. L'attribut *name* de cette classe contiend le chemin d'accès à la caractéristique dont on désire obtenir la valeur. Si on désire prendre la valeur d'une des caractéristiques du noeud sur lequel l'utilisateur a cliqué, il suffit de spécifier le nom de celle-ci. Si par contre on désire accéder à un autre noeud, on peut le faire soit en absolu, soit en relatif par rapport au noeud sur lequel on a cliqué. En absolu, le chemin d'accès à une caractéristique a la forme : /TOW/<nom_vue>/<chemin>/<caractéristique>. Détaillons les différents éléments de ce chemin d'accès :

<nom_vue>	:	nom de la vue dans laquelle se trouve le noeud ;
<chemin>	:	chemin complet pour accéder au noeud. Il est constitué d'une suite de noms de noeuds séparés par le caractère '/'. Le dernier noeud est le noeud que l'on désire atteindre ;
<caractéristique>	:	nom de la caractéristique dont on désire obtenir la valeur.

En relatif, le chemin d'accès ne commence pas par '/' et on utilise simplement les deux caractères '..' afin de remonter d'un cran dans l'arborescence.

Dans certains cas, la valeur n'est pas connue du *WebTree*, mais celui-ci peut interroger un service *OpenMaster* prédéfini afin de récupérer un ensemble de caractéristiques qu'il peut utiliser pour calculer la valeur du paramètre. Ceci est représenté par la classe *Request*. Une *Request* comporte deux paramètres d'exécution de la requête -*parameters*- et une série de filtres -*filters*. Les paramètres ayant pour but de préciser la demande effectuée auprès du

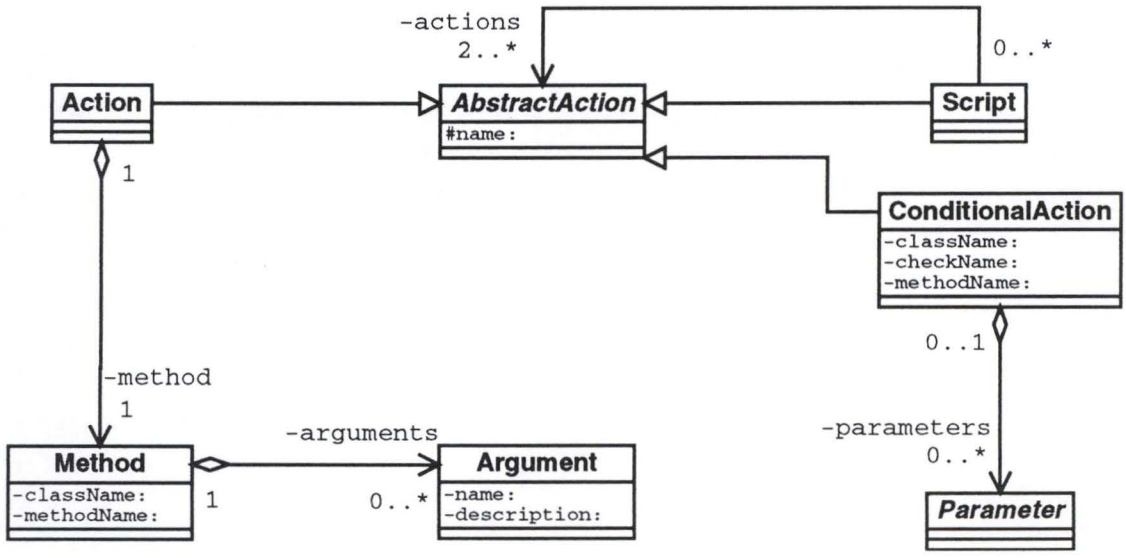


FIG. 2.5 – L'action

service *OpenMaster*, ils doivent également pouvoir être calculés à partir des caractéristiques du noeud. Une fois la requête effectuée, le service renvoie un ensemble de caractéristiques permettant au *WebTree* de créer logiquement un noeud (sans aucune représentation dans l'arbre). Le *WebTree* peut alors utiliser ce noeud pour calculer la ou les valeurs désirées pour le paramètre de l'action. Les filtres de la requête déterminent la manière dont le *WebTree* doit calculer ces valeurs.

Il peut arriver également que la valeur à passer à l'action soit la concaténation de plusieurs valeurs pouvant elles-mêmes être calculées d'après les caractéristiques du noeud. Nous utilisons alors le concept de *Concatenation*. Une concaténation est constituée d'au moins deux éléments.

Enfin, certaines actions demandent des paramètres de type vecteur d'éléments. Ceci est modélisé par le concept de *Vector*.

2.1.2.4 Les actions

Comme on peut s'en rendre compte sur le schéma de la figure 2.5, le concept d'action est un peu plus complexe. Nous commencerons par expliquer la classe *AbstractAction*. Cette classe abstraite a pour but de représenter l'ensemble des actions pouvant être exécutées lorsque l'utilisateur clique sur un élément du menu déroulant associé à un noeud quelconque. Toute action pouvant être définie dans le *WebTree* doit posséder un nom. Ceci est matérialisé par l'attribut *name* de cette classe.

Un premier type d'action pouvant être exécuté est l'appel à une méthode quelconque d'une classe Java. Ce type est représenté par la classe *Action* qui hérite de *AbstractAction*.

La méthode appelée est quant à elle modélisée par les classes *Method* et *Argument*. Les attributs *className* et *methodName* représentent respectivement le nom de la classe qu'il faut instancier et le nom de la méthode de cette classe qu'il faut appeler pour exécuter l'action. La classe *Argument* représente la définition d'un argument à passer lors de l'appel. Cette définition consiste en le nom de l'argument et une brève description. Il ne peut évidemment y avoir deux arguments de même nom pour une même méthode. L'affectation d'un nom à chaque argument permet de pouvoir passer les arguments dans un ordre quelconque. De plus, si un argument est manquant, le *WebTree* peut demander à l'utilisateur d'entrer une valeur pour cet argument. L'attribut *description* est quant à lui affiché pour aider l'utilisateur dans le choix d'une valeur.

Un deuxième type d'action est le *Script*. Le but de ce type d'action, qui hérite également de la classe *AbstractAction*, est de pouvoir exécuter, dans un ordre donné, une suite d'actions préexistantes.

Enfin, le troisième et dernier type d'action est représenté par la classe *ConditionalAction*. Il s'agit ici de répondre au besoin d'action conditionnelle (cf. 1.2.2.3, page 28). Les attributs *className*, *checkName* et *methodName* de la classe *ConditionalAction* représentent respectivement le nom de la classe et de la méthode à appeler pour vérifier si l'action doit être affichée dans le menu déroulant associé à un noeud et le nom de la méthode à appeler lors de l'exécution de l'action. De plus, une *ConditionalAction* possède un ensemble de *Parameters* permettant de calculer la valeur des paramètres de l'action au moment de son exécution.

2.1.2.5 Le *NodeType*

Le *NodeType*, représenté sur le schéma de la figure 2.6, a pour but de décrire un type de noeuds. Ce type de noeuds rassemble un ensemble de propriétés communes à un groupe de noeuds. Parmi ces propriétés, nous avons un ensemble de caractéristiques (classe *Characteristic*), les éléments qui composent le menu associé à ces noeuds (classes *Menu* et *MenuItem*), une éventuelle action à exécuter automatiquement lors de la création d'un noeud de ce type (classe *AbstractAction*) ainsi que les paramètres d'exécution de cette action automatique. Chaque *NodeType* possède au moins une caractéristique dont l'attribut *name* vaut "name" : il s'agit du nom du *NodeType*. De plus, chaque *NodeType* référence un *NodeType* considéré comme parent. Comme pour le concept de *ClassType*, l'ensemble des *NodeTypes* forme une arborescence. Le seul *NodeType* à ne pas avoir de parent est donc le *NodeType* racine, appelé également "root".

La classe *MenuItem* hérite de *Menu* et représente un élément du menu. Chaque *Menu* possède un nom et contient un ensemble de *Menus* ou de *MenuItems*. Un menu associé à un noeud peut ainsi contenir des sous-menus. Chaque *MenuItem* référence une action et possède un ensemble de paramètres d'exécution de cette action.

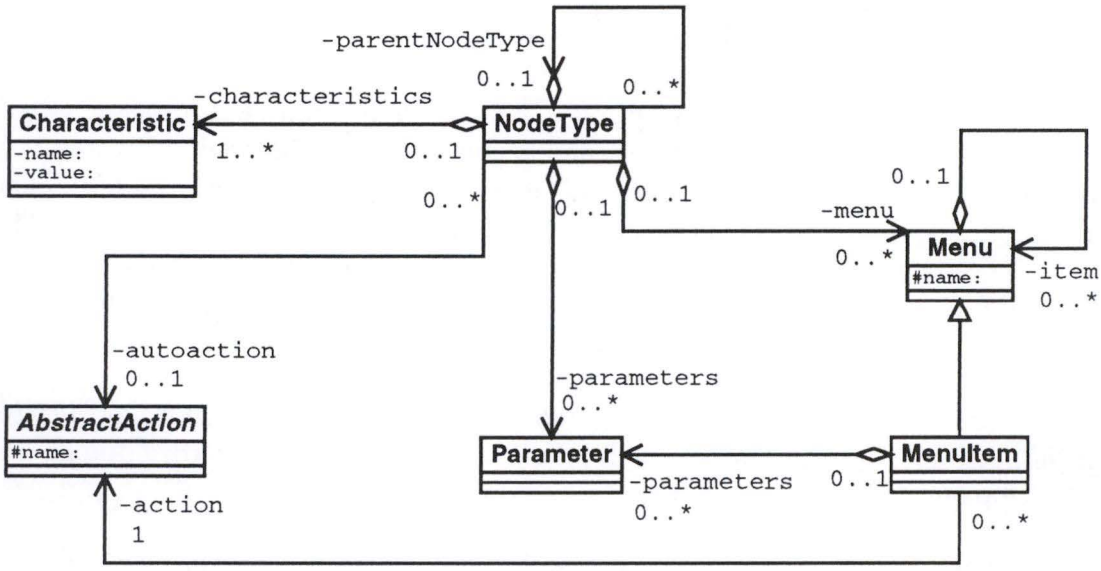


FIG. 2.6 – Le *NodeType*

2.1.2.6 La *Policy*

Le concept de *Policy*, décrit par le schéma de la figure 2.7, représente une règle de *Drag&Drop* entre deux noeuds. Une règle se base sur le *ClassType* du noeud source du *Drag&Drop* et sur le *ClassType* du noeud cible. Lors du *Drag&Drop*, une action (classe *AbstractAction*) est exécutée avec un certain nombre de paramètres (classe *Parameter*). Cette action peut être, par exemple, de réaliser une copie du noeud, ou encore de le déplacer.

Comme nous l'avons explicité plus haut, un *ClassType* peut posséder des sous-*ClassTypes*. Or, il peut être intéressant de spécifier pour une règle qu'elle peut non seulement avoir pour source le *ClassType* associé, mais également n'importe lequel de ses fils. Il en est de même pour le *ClassType* de destination. C'est le but des attributs *sourceType* et *destinationType*.

Ce concept permet notamment de répondre aux besoins de *Configuration Manager* (cf. 1.2.2.4, page 28). En effet, prenons un noeud de l'arbre. Admettons que ce noeud, nommé "Créer utilisateur Unix", représente l'action *Configuration Manager* de créer un utilisateur sur une machine Unix. Ce noeud aurait le *ClassType* "actionConfigManager". Prenons maintenant un autre noeud représentant une machine Unix appelée "Alice". Ce noeud aurait pour *ClassType* "machineUnix". De plus, admettons que parmi les actions définies dans le *WebTree*, une déclenche l'exécution d'un script qui appel l'action *Configuration Manager* que nous venons d'expliquer. Cette action du *WebTree* s'appelerait "CreateUser" et prendrait deux arguments : le nom de l'ordinateur et le nom de l'utilisateur à créer. Enfin,

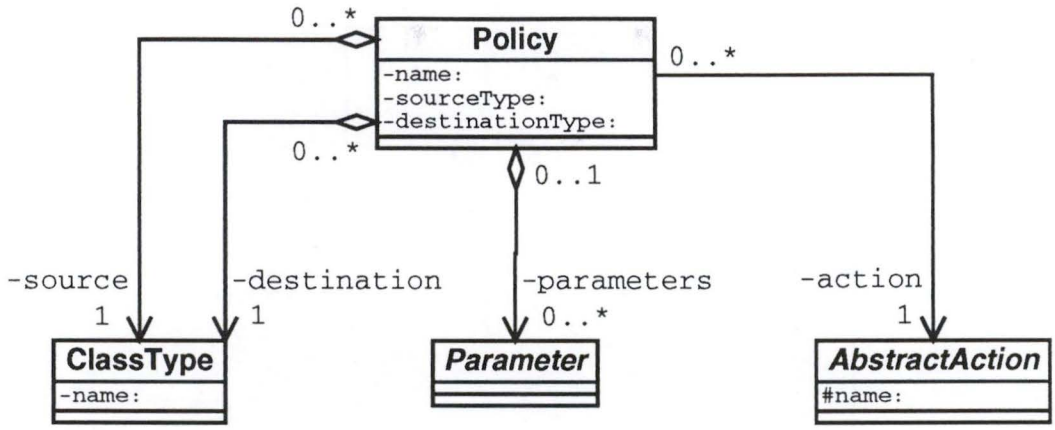


FIG. 2.7 – La *Policy*

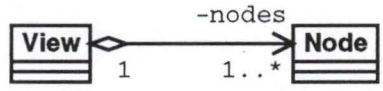


FIG. 2.8 – La *View*

créons une règle de *Drag&Drop* qui aurait pour source le *ClassType* "actionConfigManager", pour destination le *ClassType* "machineUnix" et qui aurait "CreateUser" pour action. Si l'utilisateur glisse le noeud "Créer utilisateur Unix" sur le noeud "Alice", le *WebTree* demandera le nom de l'utilisateur à créer puis exécutera l'action *Configuration Manager* de création de l'utilisateur sur la machine "Alice".

2.1.2.7 La *View*

Comme on peut le voir sur le schéma de la figure 2.8, une *View* est constituée d'un ensemble de noeuds. Toute vue comporte au minimum un noeud racine. Le nom de la vue correspond au nom du noeud racine. De plus, tout noeud doit appartenir à une vue.

2.1.2.8 Le *Package*

Enfin, le dernier concept à présenter est celui de *Package* dont le schéma est présenté à la figure 2.9. Un *Package* est constitué d'un ensemble de *ClassTypes*, de *NodeTypes*, d'actions et de *Policys*. De plus, un *Package* peut avoir des sous-*Packages*. De cette façon, différentes équipes de développements ou différents administrateurs d'*OpenMaster* peuvent créer ou éditer les *Packages* propres à leurs besoins sans risquer d'interférer avec d'autres équipes ou administrateurs qui éditeraient les leurs.

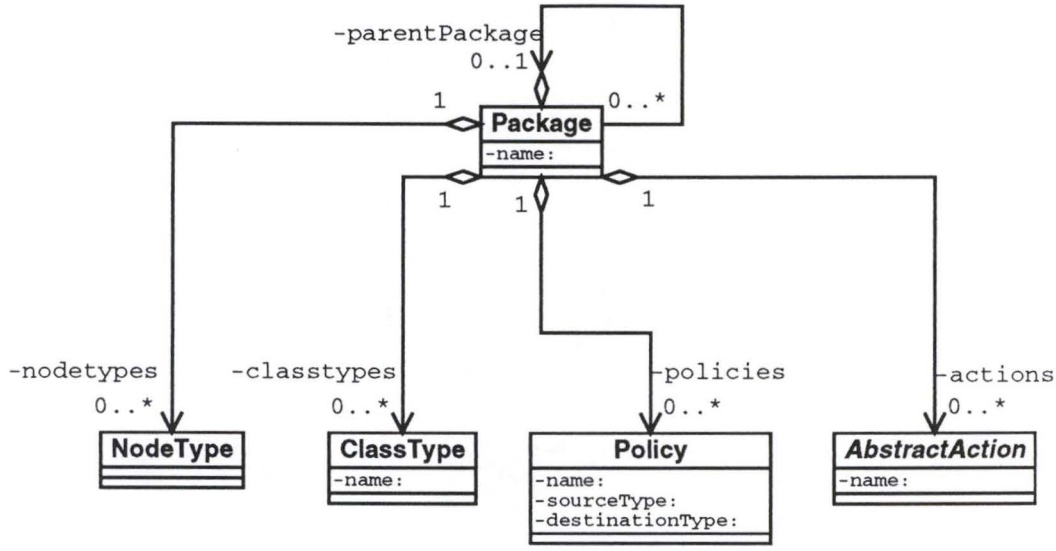


FIG. 2.9 – Le *Package*

2.2 L'interface

Maintenant que nous avons vu les concepts principaux du *WebTree*, nous pouvons examiner plus en avant le fonctionnement de l'application. Pour ce faire, nous allons nous mettre à la place d'un utilisateur et expliquer le déroulement d'une utilisation du *WebTree*. Ce déroulement sera un peu particulier dans le sens où un utilisateur final ne réalisera probablement pas tout ce que nous allons expliquer. En effet, en général, l'utilisateur se limitera à travailler avec une ou deux vues. Il ne fera par exemple probablement pas d'édition : ceci sera plutôt réalisé par un des administrateurs d'*OpenMaster*. En fait, on peut dire que notre utilisateur joue plusieurs rôles.

2.2.1 Le lancement du *WebTree*

Lorsque nous ouvrons le domaine *WebTop* appelé "applications", l'application générique *WebTree* est lancée automatiquement. Le *WebTree* ouvre alors la ou les vues ouvertes lorsque nous avons quitté *OpenMaster*. Nous admettrons que nous avions fermé toutes les vues avant de quitter l'application. Le *WebTree* a alors l'apparence de la figure 2.10. Toutes les images que nous montrerons durant cette présentation ne représentent que le *WebTree*. Il ne faut néanmoins pas oublier que celui-ci est normalement exécuté à l'intérieur de l'applet *WebTop*. Comme nous pouvons le voir sur cette image, nous avons trois menus à notre disposition : "Vues", "Edition" et "Fenêtre". Décrivons les différents éléments qui composent chacun de ces menus.

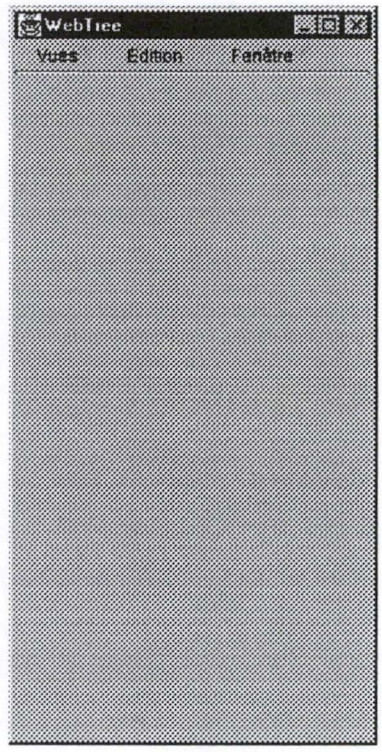


FIG. 2.10 – Lancement du *WebTree*

Le menu "Vues"

Le menu "Vues" propose les différentes actions qui peuvent être effectuées sur les vues. Il possède six items : "Nouvelle" pour créer une nouvelle vue et l'éditer, "Ouvrir" pour charger une vue existante, "Editer" pour ouvrir une vue existante en mode d'édition (nous expliquerons plus loin ce que nous entendons par *mode d'édition*), "Sauver" pour enregistrer les modifications effectuées dans une vue ouverte en mode d'édition, "Fermer" pour fermer une vue et "Supprimer" pour effacer la vue.

Le menu "Edition"

Outre la possibilité de rechercher un noeud dans une vue, le menu "Edition" offre les possibilités classiques de "couper", "copier" et "coller" un noeud.

Le menu "Fenêtre"

Le *WebTree* offre la possibilité d'afficher simultanément deux vues à l'écran. Le menu "Fenêtre" permet de choisir entre les deux modes d'affichage : "Une vue" ou "Deux vues".

Certains éléments du menu peuvent être grisés. En effet, il est possible que l'utilisateur

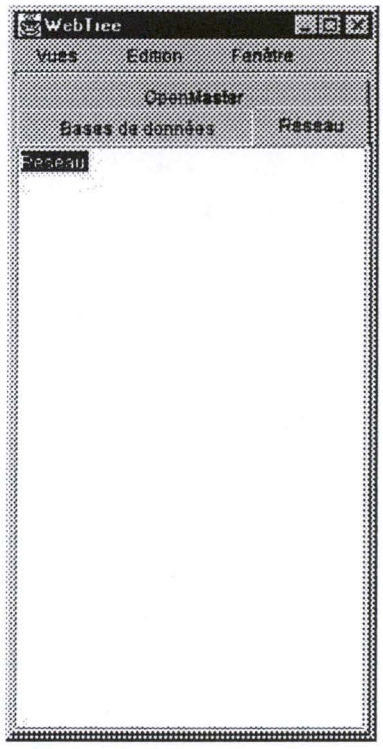


FIG. 2.11 – Le *WebTree* en mode "Une vue"

possède un rôle qui par exemple ne lui permette pas d'éditer une vue. Toutes les entrées de menu relatives à l'édition sont alors inaccessible à l'utilisateur. Bien que dans notre description, nous considérons avoir tous les droits d'accès, nous expliquerons à quel moment le *WebTree* vérifie ces droits afin de répondre au besoin de rôles expliqué dans la partie 1.2.2.2, page 27.

2.2.2 Le chargement d'une vue

Choisissons l'item "Ouvrir" du menu "Vues". Le *WebTree* propose alors la liste des vues existantes. Pour composer cette liste, il prend toutes les vues qui existent et pour chacune vérifie auprès du gestionnaire de rôles[2] si le rôle que nous avons nous permet d'y accéder en lecture. Si tel est le cas, la vue nous est proposée. Sélectionnons par exemple la vue "OpenMaster", vue de base créée par le projet *AGOV*. Le *WebTree* charge alors cette vue ainsi que tous les *Packages* qu'elle utilise. La vue affichée est composée de deux parties : des onglets et une arborescence de noeuds. Si plusieurs vues sont chargées, les onglets permettent de sélectionner laquelle afficher. Admettons que nous chargions deux autres vues : "Bases de données" puis "Réseau". La figure 2.11 montre ce que nous avons alors à l'écran. Si nous double-cliquons sur le noeud "Réseau", nous affichons ses fils. Sélectionnons

[2]Le gestionnaire de rôles est un composant du *WebTop* auprès duquel les différentes applications peuvent savoir si l'utilisateur courant peut accéder à une ressource donnée.

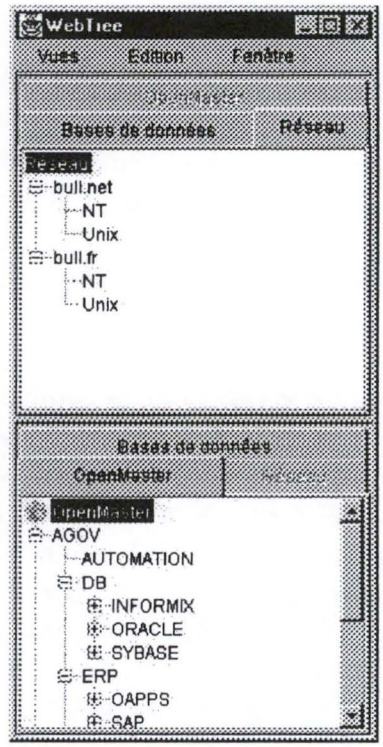


FIG. 2.12 – Le *WebTree* en mode "Deux vues"

maintenant l'item "Deux vues" du menu fenêtre. Le *WebTree* se coupe alors en deux et nous présente deux vues distinctes. Il n'est pas possible d'afficher deux fois la même vue. Cela aurait en effet posé un certain nombre de problèmes au niveau de l'implémentation afin de gérer la cohérence entre les deux "vues graphiques" affichées. Cela n'étant pas nécessaire aux yeux de notre commanditaire, nous avons décidé que ce n'était pas possible dans la version que nous développions. La figure 2.12 nous montre l'affichage simultané des vues "Réseau" et "OpenMaster", quelques noeuds de ces vues ayant été ouverts afin de montrer leurs fils. Si nous laissons la souris quelques secondes sur un noeud, le *WebTree* vérifie si le noeud possède la caractéristique "message". Si tel est le cas, la valeur de cette caractéristique est affichée dans une bulle d'aide. Repassons maintenant en mode "une vue".

Pour afficher les différents noeuds, le *WebTree* se base sur les caractéristiques du noeud. Certaines caractéristiques telles que "icônes", "badState", "goodState", etc. sont ainsi utilisées afin d'afficher une icône en plus du nom du noeud, nom qui correspond à la caractéristique "name" du noeud. Bien entendu, ces caractéristiques ne sont pas toujours présentes et ne sont pas forcément utilisées. Les caractéristiques "badState", "goodState", etc. ne sont par exemple utilisées que si l'utilisateur a demandé que le noeud soit animé. Dans ce cas, le *WebTree* met alors à jour la caractéristique "state" en faisant appel à une autre application du *WebTop*. Suivant la valeur de cette caractéristique, l'icône référencée par "badState", "goodState", etc. est affichée.

2.2.3 L'exécution d'une action

Cliquons maintenant sur un noeud avec le bouton droit de la souris. Le *WebTree* nous affiche alors un menu déroulant. Afin de constituer ce menu déroulant, le *WebTree* récupère le nom du *NodeType* dans les caractéristiques du noeud. Le *WebTree* utilise alors le gestionnaire de rôles afin de vérifier pour chaque menu et entrée de menu configurés dans le *NodeType* et ses parents que l'utilisateur y a accès. Si tel n'est pas le cas, l'entrée de menu n'est pas ajoutée au menu déroulant. Le *WebTree* interroge également les différentes *ConditionalActions* chargées pour savoir si elles désirent être ajoutées au menu déroulant. Le menu déroulant ainsi constitué est alors montré à l'utilisateur. Celui-ci ne peut donc voir que les entrées auxquelles il a accès. Si nous cliquons maintenant sur une des actions proposées, le *WebTree* l'exécute en utilisant les caractéristiques du noeud pour calculer les paramètres éventuels à passer à l'action. Il est possible à certaines actions de retourner un ensemble de valeurs qui permettent au *WebTree* de créer un sous-noeud en dessous du noeud à partir duquel l'action a été lancée. Si l'action demandée était par exemple le lancement d'une application, l'action peut retourner des valeurs comme l'identifiant au sein du *WebTop* de l'application lancée. Un sous-noeud représentant cette application est créé. Il sera alors possible, à partir du *WebTree*, de fermer cette application, de la rendre visible, etc.

2.2.4 L'édition

Cliquons maintenant sur l'élément "Editer" du menu "Vues". Parmi les différentes vues que nous pouvons éditer, une est un peu particulière : il s'agit de la vue "Packages". Cette vue permet à l'utilisateur de créer et d'éditer les différents *Package*s que le *WebTree* utilise lorsqu'il affiche une vue. Ouvrons donc cette vue en édition. Affichons ensuite les sous-noeuds de "Packages". Nous voyons apparaître les *Package*s : "DB", "Edition", "ERP", "Exemple", "OS" et "TP". Nous allons nous intéresser à deux *Package*s en particulier : les packages "Exemple" et "Edition", le premier afin de présenter l'édition d'un *Package*, le deuxième car il s'agit d'un *Package* essentiel pour l'édition.

Ouvrons quelques sous-noeuds du *Package* "Exemple" afin d'obtenir l'arborescence montrée par l'image 2.13. On remarquera tout d'abord les quatre sous-noeuds directs du noeud "Exemple" : "actions", "classtypes", "nodetypes" et "policies". Ils correspondent aux différentes associations montrées lorsque nous avons expliqué le concept de *Package* (cf. 2.1.2.8, page 43). Il en est de même pour leurs sous-noeuds. Ainsi, on peut par exemple voir que l'attribut "name" de l'action "Action2" a la valeur "Action2", que l'attribut "className" de la "method" associée à cette action vaut "ism.TOW.actions.Action2", etc. De même, pour le *NodeType* "NodeType3", on remarquera tout d'abord que celui-ci a pour "parentNodeType" le *NodeType* "NodeType1" du *Package* "Exemple". On voit également qu'une icône lui est associée : "/Images/Icon.gif"'. De plus, lorsqu'un noeud de *NodeType* "NodeType3" est créé, l'action "Action3" est automatiquement exécutée avec comme paramètre "vnodes", un vecteur contenant un seul élément : le nom du noeud qui vient d'être créé.

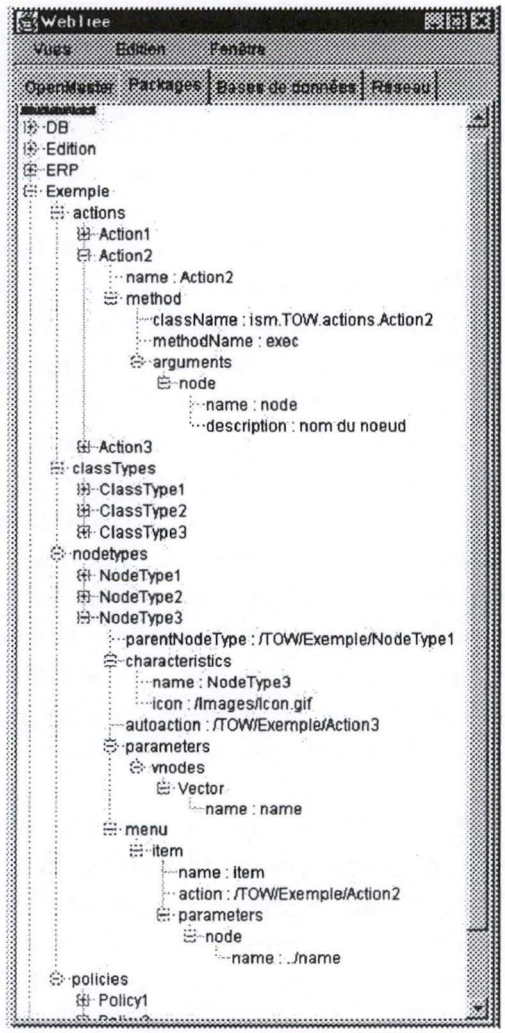


FIG. 2.13 – L'édition de *Packages* du *WebTree*

Enfin, le *NodeType* "NodeType3" ajoute un élément (appelé "item") au menu du *NodeType* "NodeType1" dont il hérite. Lorsque l'utilisateur clique sur cet élément, l'action "Action2" est exécutée avec pour paramètre "node", le nom du noeud père du noeud sur lequel on a cliqué. Le même principe est utilisé pour les *ClassTypes* et les *Policys*.

Si nous cliquons sur le noeud "icon : /Images/Icon.gif" dans les caractéristiques du *NodeType* "NodeType3", il nous est possible de modifier directement tant le nom que la valeur de cette caractéristique. Par contre, pour le "parentNodeType", il est bien entendu impossible de modifier le nom de cette propriété. De même, le noeud "NodeType3" n'est quant à lui pas modifiable. Par contre, si on modifie la valeur de sa caractéristique "name", son nom est également modifié. Nous pourrions continuer ainsi pour chacun des noeuds de la vue mais expliquons plutôt le mécanisme général utilisé pour l'édition d'un noeud.

Le mécanisme d'édition au sein du *WebTree* est en fait basé sur le *Package* appelé "Edition". Ce *Package* définit un ensemble d'actions, de *ClassTypes*, de *NodeTypes*, et de *Policys* qui déterminent le fonctionnement de l'édition. A l'exception du fait que certains champs sont éditables, le *WebTree* considère en effet une vue en édition comme une vue *normale*. Toutefois, lorsqu'une vue est chargée afin d'être éditée, les *ClassTypes* et *NodeTypes* affectés aux différents noeuds ne sont pas ceux enregistrés dans la configuration de la vue mais bien des *ClassTypes* et *NodeTypes* du *Package* "Edition". Un *ClassType* et un *NodeType* d'édition d'un noeud sont donc affectés à chacun des noeuds de la vue. Ce *NodeType* possède une action automatique qui va créer des sous-noeuds afin de représenter les différentes caractéristiques du noeud. Il existe ainsi toute une série de *ClassTypes* et de *NodeTypes* d'édition. Les différents noeuds représentant une des associations ou une des propriétés des différentes classes que nous avons présentées dans la partie consacrée aux concepts (cf. 2.1.2) possèdent donc chacun un *ClassType* et un *NodeType* spécifique afin de déterminer les possibilités d'édition offertes à l'utilisateur pour tel ou tel noeud.

Revenons au *Package* "Exemple" que nous sommes en train d'éditer afin de montrer ce que ces actions, *ClassTypes* et *NodeTypes* d'édition permettent de faire. Si nous cliquons avec le bouton droit de la souris sur le noeud "characteristics" du *NodeType* "NodeType3", le menu déroulant qui apparaît nous propose d'ajouter une nouvelle caractéristique. Le menu déroulant associé aux noeuds "parameters" nous propose quant à lui d'ajouter un nouveau paramètre de type "Concatenation", "Value", "Variable", "Vector" ou encore "Request". Les références à d'autres objets (*actions*, *ClassTypes* ou *NodeTypes*) peuvent être modifiées de deux manières : soit l'utilisateur clique sur le noeud et édite la référence, soit il sélectionne l'objet à référencer parmi les différents *Packages* et le glisse sur le noeud représentant la référence. Ainsi, si nous désirons changer l'action automatique de notre *NodeType* pour que l'action "Action1" soit exécutée, nous pouvons soit cliquer sur le noeud "autoAction : /TOW/Exemple/Action3" et modifier "/TOW/Exemple/Action3" en "/TOW/Exemple/Action1", soit sélectionner le noeud "Action1" dans les actions et le glisser sur le noeud "autoAction : /TOW/Exemple/Action3". Il ne sera par contre pas possible de glisser l'action "Action1" sur le noeud "parentNodeType : /TOW/Exemple/NodeType1".

Ces deux noeuds possèdent en effet des *ClassTypes* différents et il n'existe aucune *Policy* permettant le *Drag&Drop* entre ceux-ci.

Ce système permet également d'ajouter des actions aux différents menus d'édition afin, par exemple, d'afficher une fenêtre complète d'édition d'un noeud en partie droite du *WebTop*. Le système d'édition direct au sein du *WebTree* nous paraît en effet assez convivial si quelques caractéristiques de noeuds doivent être modifiées. Par contre si un administrateur doit créer complètement une vue comportant de nombreux noeuds, il nous paraît plus convivial d'afficher en partie droite une fenêtre lui permettant d'entrer directement toutes les caractéristiques d'un noeud. *AGOV* pourrait donc créer ses propres interfaces de définition de noeuds, *ClassTypes*, *NodeTypes*, etc. et utiliser l'*API* que le *WebTree* doit offrir afin de créer ces objets dans l'arbre.

Enfin, il nous reste à préciser que toute modification réalisée dans une vue n'est pas prise en compte directement. En effet, si une vue que nous appellerons vue de travail est chargée dans le *WebTree* et que nous réouvrons cette même vue en édition, après avoir enregistré les modifications effectuées dans la vue en édition, il faudrait fermer et réouvrir la vue de travail pour tenir compte des changements. Les *Packages* chargés en mémoire le restant jusqu'à ce que l'on quitte le *WebTree*, la seule façon de tenir compte de changements réalisés dans un *Package* est de quitter et de relancer le *WebTop*.

2.3 Conclusion

Nous voici arrivé au terme de ce second chapitre qui nous a permis de nous familiariser avec les différents concepts du *WebTree* ainsi qu'avec les diverses possibilités qu'il offre à l'utilisateur. Comme nous avons pu le voir, ces différents concepts et possibilités répondent déjà à de nombreux besoins exprimés dans le chapitre précédent. Nous avons ainsi pu montrer comment nous avons répondu aux besoins principaux d'*AGOV* depuis celui d'un arbre pour représenter les domaines *AGOV* à celui d'édition en passant par les caractéristiques modulables d'un noeud, les actions, les menus contextuels, l'animation, le concept de vue, et l'application générique. Nous avons également pu remarquer que certains besoins externes ont également été pris en compte. Nous parlons ainsi des besoins généraux de gestion de rôles, et d'actions conditionnelles mais également de ceux de l'application *Configuration Manager*.

Maintenant que ces concepts ont été décrits, nous pouvons aborder les différents choix technologiques que nous avons dû effectuer durant le développement du *WebTree* afin de mettre en place tous les concepts et de répondre aux derniers besoins. Ce sera l'objet du chapitre suivant.

Chapitre 3

Les technologies du *WebTree*

Maintenant que nous avons vu les différentes interactions possibles entre l'utilisateur et le *WebTree*, nous allons entrer un peu plus au coeur de celui-ci afin d'examiner les différentes technologies sur lesquelles il s'appuie. Durant le développement, plusieurs choix technologiques ont dû être effectués. En effet, bien souvent, plusieurs technologies étaient utilisables et nous avons dû choisir laquelle convenait le mieux pour répondre à nos besoins. A d'autres moments par contre nous n'avions pas le choix, nous étions forcé d'utiliser telle ou telle technologie afin notamment de pouvoir nous intégrer dans l'application *OpenMaster*. Ce fut le cas pour Java, Swing et *XML*.

Nous n'expliciterons pas complètement les différentes technologies, nous nous limiterons à les décrire brièvement et à exposer leurs caractéristiques principales. Nous montrerons également pourquoi nous avons écarté certaines technologies et comment nous avons utilisé les autres.

Nous commencerons par présenter le langage que nous avons utilisé pour développer le *WebTree*, à savoir Java. Nous décrirons également Swing, le nouveau *toolkit* graphique de Sun. Nous essayerons de montrer en quelques mots les avantages et les problèmes posés par ces deux outils. Ensuite, nous aborderons une partie importante consacrée aux fichiers de configuration et plus précisément au nouveau langage de structuration de documents : *XML*. Ceci nous conduira à la recherche d'un métamodèle de document pour les fichiers de configuration. Il s'agira du dernier point que nous aborderons dans ce chapitre.

3.1 Java et Swing

La première technologie dont nous allons parler est Java et plus précisément Swing. Nous ne parlerons en effet pas beaucoup de Java que nous considérons comme connu. Expliquons simplement pourquoi nous avons dû l'utiliser. Comme nous l'avons dit, le *WebTree* est un composant de l'application *OpenMaster* et plus précisément du *WebTop*. Or, ce dernier doit pouvoir être exécuté dans un navigateur *WWW* tant sous Windows (95, 98

et NT) que sous Unix. Il était donc nécessaire d'utiliser un langage portable pouvant être exécuté dans un navigateur. C'est pourquoi Java fut choisi.

Swing est un nouveau composant de Java destiné à fournir une nouvelle interface graphique simplifiant le développement de fenêtres d'applications. On entend par fenêtres d'applications aussi bien les fenêtres ou boîtes de dialogue elles-mêmes que les menus, barres d'outils, boutons, etc. qui peuvent la composer. Swing fait en réalité partie d'une nouvelle librairie de classes Java appelée les *Java Foundation Classes* ou *JFC*. Cette librairie fournit d'autres possibilités telles que le *Drag&Drop* entre applications.

Examinons maintenant les nouvelles fonctionnalités apportées par Swing. Nous montrerons ensuite quelques problèmes ou contraintes dus à l'utilisation de ce *toolkit*.

3.1.1 Les nouvelles fonctionnalités de Swing

Une des plus importantes fonctionnalités offertes dans Swing est le *pluggable look and feel*. Il s'agit en fait de pouvoir modifier dynamiquement l'apparence et le comportement de tous les composants graphiques Swing. L'apparence des applications utilisant les anciens composants appelés *Abstract Windowing Toolkit* (ou *AWT*) dépend de la plate-forme où est exécutée l'application. Avec Swing, ceci n'est plus vrai. Différents *look and feel* sont fournis : un *look and feel* Windows, un *look and feel* Motif (interface graphique Unix), un *look and feel* Java et un *look and feel* appelé *Multiplex*. Ce dernier a pour but d'offrir la possibilité d'ajouter quelques fonctionnalités à une interface graphique Swing sans pour autant devoir créer un nouveau *look and feel*. Par *look and feel*, on entend aussi bien la représentation graphique des composants (couleur des composants, apparence du bord des fenêtres ou des menus, etc.) que la manière d'interagir avec eux (simple clic de souris, double clic, etc.). Chacun de ces *look and feel* peut être utilisé indépendamment de la plate-forme où l'application s'exécute. Prenons un exemple afin d'expliquer cette possibilité.

Admettons qu'une société développe une application Java destinée à des utilisateurs travaillant dans un environnement hétérogène. Certains de ces utilisateurs travaillent donc sous Windows tandis que d'autres utilisent Unix. Non seulement les différents composants graphiques sont parfois représentés différemment sous Windows et sous Unix, mais en plus la manière d'interagir avec eux peut également être différente. Un utilisateur habitué à l'interface Windows pourrait avoir des difficultés à utiliser l'application s'il était contraint à un moment donné de devoir l'exécuter sous Unix. Il peut donc être intéressant d'offrir à chacun de ces groupes d'utilisateurs une interface correspondant à celle qu'ils ont l'habitude d'utiliser et ce quelle que soit la plate-forme où est exécutée l'application. Ceci est possible grâce à Swing. Il aurait également été possible de décider de créer une nouvelle interface spécifique à l'application et d'utiliser celle-ci sur les deux plates-formes. En effet, les développeurs ont la possibilité de créer de toutes nouvelles interfaces et d'appliquer celles-ci aux différents composants Swing.

De plus, les composants *AWT* sont considérés comme "heavyweight" c'est-à-dire qu'ils utilisent du code spécifique (généralement en C ou C++) à la plate-forme où ils sont exécutés. Par opposition, les composants Swing sont "leightweight" puisqu'ils n'utilisent aucun code spécifique à l'environnement. Les composants "leightweight" sont généralement plus légers et nécessitent moins de code pour leur utilisation. Certains composants Java ne pouvant être utilisés qu'avec des composants "leightweight", le fait d'avoir de tels composants graphiques est donc très intéressant.

Une autre fonctionnalité intéressante est le *Drag&Drop*. Dans certains cas, celui-ci était déjà possible avec les composants *AWT*. Une restriction majeure était le fait qu'il n'était pas possible de faire du *Drag&Drop* entre fenêtres distinctes ou entre *beans*. Grâce aux composants Swing et plus précisement grâce aux *JFC*, cela est maintenant possible. De plus, la manière de gérer le *Drag&Drop* a été revue afin d'être plus facilement utilisable pour l'application tout en augmentant les possibiltés.

Enfin, un autre intérêt de Swing est une augmentation de l'accessibilité aux composants. On entend par là le fait de pouvoir interagir avec les composants Swing en utilisant des outils matériels et logiciels spécifiques tels que des écrans tactiles, etc.

3.1.2 Quelques problèmes d'utilisation

Bien que Swing soit apparu au printemps 1998, il n'a pas toujours été possible de l'utiliser directement. En effet, Swing nécessite l'utilisation de la version 1.1.5 (ou ultérieure) de la *JDK*[1]. Or, SUN ne met à la disposition des développeurs que les *JDK* pour les environnements Sun-Solaris et Windows. Les autres fournisseurs de systèmes d'exploitation doivent porter eux-mêmes les nouvelles JDK pour leurs environnements. C'est ainsi, par exemple, que bien que la *JDK 1.2* soit disponible sous Sun-Solaris et Windows, elle ne l'est pas encore dans l'environnement *AIX* de chez IBM. Les développeurs travaillant sur des systèmes où la version 1.1.5 n'était pas encore disponible à l'époque étaient donc contraints de continuer à utiliser les anciens composants graphiques de Java : *AWT*.

Un autre problème est que l'utilisation mixte de composants Swing et *AWT* ne fonctionne pas toujours correctement. En effet, pour que les composants Swing réagissent convenablement, ils ne peuvent être inclus dans des composants *AWT*. Ainsi, un bouton Swing peut ne pas s'afficher s'il est positionné dans une fenêtre *AWT*. Bien que l'inverse ne pose par contre pratiquement aucun problème, il arrive parfois qu'il y ait quelques erreurs d'alignement entre les composants. La migration d'une application importante vers Swing peut donc être problématique. Il faut en effet pouvoir migrer l'application dans son ensemble et pas simplement un composant puis l'autre.

[1] JDK : *Java Development Kit.*

3.2 *XML*

Nous allons maintenant aborder la deuxième partie de ce chapitre consacrée à *XML*. Mais avant de présenter *XML*, revenons quelques instants aux besoins exprimés dans le chapitre 1 et plus particulièrement au besoin d'un format de fichier unique (cf. 1.2.2.1, page 26). On peut considérer un fichier de configuration comme un document contenant un ensemble d'éléments définissant la configuration d'une application. On peut alors découper la lecture (ou la sauvegarde) de la configuration en deux parties : la lecture du document et l'interprétation des éléments de ce document. La lecture du document ne dépend que du format du fichier tandis que l'interprétation des éléments dépend de l'application. Afin de répondre au besoin que nous venons de citer, il serait intéressant d'avoir une application capable d'éditer n'importe quel fichier de configuration d'*OpenMaster*. Cette application doit donc pouvoir aussi bien lire (ou écrire) le fichier qu'en interpréter le contenu. Cette interprétation ne doit pas forcément être complète mais bien suffisante pour pouvoir éditer les éléments du fichier de configuration. Nous réaliserons ce dernier point grâce à l'utilisation d'un métamodèle. Mais nous y reviendrons dans la troisième partie de ce chapitre. Il nous faut tout d'abord avoir un format unique pour les différents fichiers de configuration. Il s'agira de *XML*.

Le choix de ce format ayant été effectué avant le début de notre stage, il ne nous est pas possible d'expliquer les différentes raisons qui l'ont motivé. Nous allons par contre essayer de montrer les différents avantages de l'utilisation de *XML*. Mais expliquons tout d'abord les origines et les buts de *XML*. Or on ne peut présenter celui-ci sans parler de *SGML*. C'est donc par là que nous commencerons.

3.2.1 De *SGML* à *XML*

Le *SGML* ou *Standard Generalized Markup Language* constitue le standard général des langages de type *markup*. Ces langages sont basés sur le concept de *balise* ou *tag*. Un *tag* détermine un moyen de rendre explicite l'interprétation d'un texte. Par exemple, en français, nous utilisons de nombreux *tags* tels que la ponctuation, l'utilisation de majuscules, les espaces, la disposition des paragraphes sur une page, etc. Tous ces *tags* ont pour but de nous aider à interpréter les mots que nous lisons. C'est ce principe qu'utilisent les *markup languages* sauf que l'utilisation des *tags* ne se limite pas à la ponctuation mais va jusqu'à catégoriser chaque partie du texte. Prenons l'exemple du plus connu des *markup language* : le *HTML*. Ainsi, dans un document écrit en *HTML*, la ligne

<TITLE>Titre de la page</TITLE>

permet de définir le titre de la page *HTML* comme étant "Titre de la page". En *HTML*, tous les tags valides sont prédéfinis. Il en est autrement en *SGML*. En effet, un document *SGML* est constitué de deux parties généralement contenues dans des fichiers distincts : la structure du document et son contenu. La structure du document porte le nom de

DTD ou *Document Type Definition*. Cette *DTD* définit non seulement tous les *tags* valides pour le document mais également l'ordonnancement complet des différents tags dans le document. Reprenons le *HTML* afin de montrer le fonctionnement de la *DTD*. Bien que les navigateurs soient relativement souples quant à la structure d'une page *HTML*, celle-ci est bien définie. On pourrait donc considérer qu'il existe une *DTD* implicite définissant la structure d'un document *HTML*. Néanmoins, cette structure ne variant pas d'une page à l'autre (si ce n'est entre versions successives de *HTML*), il n'est pas nécessaire de l'associer aux différentes pages, il suffit que tout le monde la respecte. Voyons quelques règles que définirait cette *DTD*. Tout d'abord, une page *HTML* doit normalement commencer par <HTML> et se terminer par </HTML>. Entre ces deux tags, nous retrouvons deux parties : l'entête (délimité par le tag <HEAD>) et le corps de page (ou <BODY>). Le titre (*tag* <TITLE>) doit se situer dans l'entête alors que les paragraphes (*tag* <P>) se mettent dans le corps de la page. De plus, certains *tags* comme <BODY> peuvent être paramétrés grâce à l'utilisation d'attributs comme "background", etc.

Dans un document *SGML*, il n'existe pas de *DTD* prédéfinie. Chaque personne qui compose un document commence par en définir la structure puis après seulement rédige le contenu. Chacun peut ainsi utiliser la structure qui convient le mieux au document qu'il désire réaliser. Ce document doit donc toujours être accompagné de sa *DTD*. Néanmoins, rien n'empêche plusieurs documents d'utiliser la même *DTD*. Un but premier du *SGML* est ainsi atteint : l'indépendance entre les documents *SGML* et les plates-formes et applications qui manipulent ces documents. En effet, la *DTD* définissant complètement la structure d'un document, toute application utilisant cette *DTD* est à même d'éditer ce document. L'apparence à donner à l'écran reste quant à elle dépendante des applications. Ainsi, les mécanismes permettant de définir la représentation d'un *tag* à l'écran ne sont pas encore bien définis.

Tout comme le *HTML*, le *Extensible Markup Language* (ou *XML*) est un sous-ensemble du *SGML* développé par le *XML Working Group* formé sous l'égide du *W3C*[2] en 1996. L'intérêt était notamment de permettre un accès plus aisé aux *markup languages* tout en gardant une grande richesse dans le langage. Les buts premiers de *XML* sont :

1. être utilisable directement sur Internet ;
2. supporter une large variété d'applications ;
3. être compatible avec le *SGML* ;
4. l'écriture de programmes traitant des documents *XML* doit être aisée ;
5. le nombre de propriétés optionnelles du *XML* doit être le plus petit possible ;
6. les documents *XML* doivent être lisibles par l'homme et relativement clairs ;
7. la conception d'un document *XML* doit pouvoir être effectuée rapidement ;
8. la conception d'un document *XML* doit être formelle et concise ;

[2]W3C : *World Wide Web Consortium.*

9. les documents *XML* doivent être faciles à créer ;
10. la concision des *tags XML* revêt une importance minime.

Une simplification importante du langage est le caractère facultatif de la *DTD*. En effet, on considère qu'un document *XML* sans *DTD* est bien formé -correct- si ses *tags* sont utilisés convenablement. On entend par là que tout *tag* ouvert doit être fermé et que l'enchâssement doit être correct.

Afin de respecter le but de facilité d'écriture de programmes qui utilisent des documents *XML*, les concepteurs de *XML* ne se sont pas uniquement intéressés à définir un nouveau *markup language* mais ont également spécifié un certain nombre de critères que doivent respecter les programmes qui manipulent les documents *XML*. La notion de *XML processor* a ainsi été définie comme un module applicatif utilisé afin de lire les documents *XML* et d'accéder à leur structure. Ce module réalise son travail sous le contrôle d'une *application*. Les *XML processors*, appelés également *parsers XML*, ont été divisés en deux catégories : *validating* et *non-validating*. La première catégorie de *parsers XML* doit vérifier que toutes les contraintes imposées tant par les déclarations situées dans la *DTD* que par les contraintes définies dans la norme *XML* sont bien respectées. L'*application* doit être notifiée du moindre écart vis-à-vis de ces contraintes. La deuxième catégorie est nettement plus légère. Ainsi, elle ne s'attache qu'à vérifier le fait que le document (*DTD* comprise) est bien formé. Le respect de la structure définie dans la *DTD* n'est quant à lui pas vérifié. De plus, il peut arriver que dans des documents relativement complexes, certaines erreurs au niveau de la forme ne soient pas repérées par ces *parsers*, ces erreurs survenant dans des parties que le *parser* n'est pas tenu de lire. D'autres différences existent quant à la manière de traiter certaines informations lues dans le document *XML*, mais ces différences n'étant pas vitales à la compréhension de ce qui suit, nous nous limiterons à signaler leur existence. La norme *XML* définit également ce que les différents *parsers* sont tenus de notifier à l'application (par exemple, les différents éléments du document et leurs attributs) et quand ils doivent le faire (respect de l'ordre des éléments, etc.).

Bien que la norme *XML* définisse de nombreuses contraintes concernant les *parsers XML*, elle ne définit aucune *API* que les applications pourraient utiliser pour manipuler ces *parsers*. Néanmoins quelques *API* sont tout de même apparues. Il s'agit principalement de *SAX* et plus récemment de *DOM* quoique, comme nous le verrons, celle-ci est d'un autre niveau. Il est assez aisé de trouver toute une série de *parsers* répondant à l'une ou l'autre (voire même parfois les deux) de ces *API*'s. Certains sont du domaine public, d'autres doivent être achetés afin de pouvoir les utiliser à des fins commerciales. La réalisation et l'optimisation d'un *parser XML* constitue une étude approfondie de la norme *XML* et un développement important afin de gérer convenablement cette norme. Il nous a donc semblé préférable d'utiliser un *parser* du domaine public respectant une des deux *API* pré-citées afin de pouvoir changer aisément de *parser* si nous en rencontrions le besoin. Certains *parsers* n'offrant pas les deux *API*'s, nous avons tout d'abord choisi l'*API* que nous allions utiliser. Une fois cette *API* choisie, il ne nous restait plus qu'à utiliser un

parser du domaine public la respectant. Nous allons donc brièvement présenter ces deux *API*'s puis nous expliquerons laquelle nous avons choisie ainsi que les raisons motivant ce choix.

3.2.2 *SAX*

Simple API for XML (ou *SAX*) est, comme son nom l'indique, une *API* que les applications peuvent utiliser pour interagir avec un *parser XML*. Bien que cette *API* ait été développée par différentes sociétés actrices dans le domaine du *XML*, elle est cependant gratuite et peut être utilisée aussi bien à des fins non commerciales que commerciales. Basée sur un concept d'événements, elle permet à une application d'être notifiée par un *parser XML* des différents éléments découverts dans un document *XML*. Les deux types de parsers peuvent bien entendu être utilisés. Losque le *parser* décide, suivant la norme *XML*, de transmettre à l'application un élément lu dans le document *XML*, il utilise une des méthodes de l'*API*. Le choix de la méthode à utiliser dépend du type d'élément à transmettre (élément de la *DTD*, ou du document, etc.). De même, un ensemble de méthodes permettent au *parser* de signifier une erreur dans le document.

Comme nous venons de le voir, l'application est prévenue au fur et à mesure des différents éléments lus dans le document *XML*. Elle doit donc elle-même organiser la conservation de ces éléments en mémoire afin de pouvoir les utiliser une fois le document lu. Le fait que l'application doive elle-même organiser ce stockage peut être intéressant. En effet, il peut ainsi être directement organisé afin de répondre au mieux aux besoins de l'application. Lorsque le *parser XML* a terminé son travail, les différents éléments lus sont donc directement et complètement exploitables par l'application.

3.2.3 *DOM*

Tout comme le *XML*, le *Document Object Model* (ou *DOM*) a, pour sa part, été spécifié au sein d'un groupe de travail du *W3C*. *DOM* ne se situe pas au même niveau que *SAX*. Il s'agit en effet d'une interface permettant aux programmes d'accéder et de mettre à jour dynamiquement le contenu, la structure et le style de documents. Ainsi, *DOM* fournit un ensemble d'objets destinés à représenter tant les documents *XML* que *HTML*, un modèle décrivant comment ces objets peuvent être combinés et une interface standard pour accéder et manipuler ces objets.

La création en mémoire des objets respectant la norme *DOM* à partir d'un document *XML* peut être réalisée de deux manières. Tout d'abord, certaines compagnies ont développé des *parsers* spécifiques à *DOM*. Lorsqu'une application fait appel à ce *parser*, celui-ci renvoie un objet représentant le document et accessible au moyen de l'interface standard définie dans *DOM*. D'autres compagnies, plus ouvertes, offrent non pas un *parser DOM*, mais plutôt un *constructeur de document DOM*. Ce constructeur agit en fait comme une

couche venant s'intercaler entre l'application et un *parser SAX* quelconque. L'application dit généralement au *constructeur de document* quel *parser* utiliser et ensuite lui demande de créer en mémoire l'objet *document* (selon la norme *DOM*) à partir d'un document *XML*. Ces normes étant bien définies, l'application peut très facilement changer de parser *XML*, de constructeur de document, ou encore des deux.

On remarque tout de suite que ce principe simplifie grandement la lecture d'un document *XML*. En effet, l'application n'a plus à gérer les différents événements définis dans *SAX*. En un ou deux appels de fonctions, un ensemble d'objets représentant le document dans son intégralité (structure et contenu) est créé en mémoire. L'application n'a alors plus qu'à utiliser ces différents objets à sa guise. De plus, l'ensemble des méthodes mises à la disposition de l'application afin de travailler sur ce document est très complet. Il est ainsi très aisé de récupérer tous les éléments représentés dans le document *XML* par tel ou tel *tag*. Il est aussi assez simple d'éditer le document afin d'ajouter des éléments représentant telle ou telle partie du document. Une fois cette édition réalisée, l'application peut également demander à enregistrer le document en mémoire dans un fichier. Bref, on peut dire que l'ensemble des fonctionnalités offertes par *DOM* est très complet.

3.2.4 Quelle *API* utiliser ?

Comme nous l'avons expliqué au début de cette partie consacrée à *XML*, cette technologie sera utilisée dans le cadre des fichiers de configuration. En ce qui concerne le *WebTree*, il s'agira principalement des fichiers décrivant les vues et les *Package*s. Prenons le cas d'un fichier décrivant une vue. Chaque noeud de la vue est donc décrit avec toutes ses caractéristiques dans le fichier. Or il est probable que certaines vues contiennent de très nombreux noeuds. Imaginons en effet une vue représentant le réseau d'une multinationale. Les machines de ce réseau se comptent par milliers. Si chaque machine est représentée par un noeud, cela fait de nombreux noeuds à stocker. Dans la description de *SAX* et de *DOM* que nous venons de réaliser, nous avons vu que *DOM* se situe plutôt entre l'application, en l'occurence le *WebTree*, et *SAX*. Essayons donc tout d'abord de voir s'il peut être intéressant pour nous de faire appel à *DOM*. Autrement dit, voyons ce qui dans notre cas est préférable : créer nous même les objets représentant le contenu du fichier de configuration ou utiliser *DOM* pour modéliser ces objets.

Comme nous l'avons vu au chapitre précédent, lorsque l'utilisateur clique sur un noeud, un menu déroulant est affiché. Vu le grand nombre possible de noeuds et de menus différents, le *WebTree* doit créer ce menu dynamiquement. Afin d'avoir le meilleur confort d'utilisation possible à ce niveau, le menu doit être affiché quasi instantanément. Or, si nous utilisons *DOM*, avant de pouvoir accéder aux caractéristiques du noeud, afin par exemple de connaître son *NodeType*, il faut tout d'abord retrouver le noeud dans le document *DOM*. Une fois le noeud retrouvé, nous pouvons alors accéder à sa caractéristique "nodetype". Il nous faut ensuite parcourir le document *DOM* représentant le *Package* contenant le *NodeType* désiré afin de trouver celui-ci. Si le *parentNodeType* de ce *NodeType* n'est

pas "/TOW/root", il faut également le rechercher, et ainsi de suite jusqu'à ce que l'on aie pu créer complètement le menu déroulant. Il apparaît donc que, bien que les différents objets *DOM* offrent de nombreuses possibilités, ils ne sont pas optimisés en fonction de nos besoins. Il est donc préférable de concevoir nos propres objets en fonction de nos besoins. Ainsi, à la création d'un noeud de l'arbre nous pourrions par exemple stocker une référence sur son *NodeType* plutôt que le nom de celui-ci. Il n'y aurait ainsi plus aucune recherche à effectuer pour créer le menu.

Bien qu'il soit donc préférable de travailler avec nos propres objets, cela ne signifie pas pour autant qu'il faille rejeter *DOM*. En effet, on pourrait toujours l'utiliser pour la lecture du fichier de configuration puis créer nos objets à partir de ceux de *DOM*. Néanmoins, ici aussi un problème se pose. Dans l'exemple de vue donné ci-dessus, on a montré qu'une seule vue pouvait contenir plusieurs milliers de noeuds. Or, si nous passons par *DOM*, cela signifie que tous ces noeuds vont être chargés une première fois en mémoire à l'ouverture de la vue puis que l'on va créer nos propres objets à partir de ceux de *DOM*. De plus, la création en mémoire d'un noeud entraîne la lecture des *Package*s non encore chargés et utilisés par ce noeud afin de pouvoir référencer les différents *NodeTypes*, *ClassTypes*, etc. Il faut également noter le fait que, vu toutes les possibilités qu'ils offrent, les objets *DOM* sont relativement gourmands en espace mémoire par rapport à nos propres objets. Or le *WebTree* n'est qu'un morceau d'une application qui nécessite déjà de très nombreuses ressources. Il est donc impératif de limiter le plus possible l'espace mémoire nécessaire à son fonctionnement. Enfin, la création des objets à partir de ceux de *DOM* entraînerait un surcroît de travail par rapport à une création "à la volée", c'est-à-dire au fur et à mesure de la réception des événements *SAX*.

Au vu des différentes raisons que nous venons de présenter, *DOM* ne sera pas utilisé par le *WebTree*. Nous utiliserons uniquement un *parser SAX* du domaine public et créerons nos propres objets en fonction des événements *SAX* reçus.

3.3 Un métamodèle

Comme nous l'avons expliqué au début du point 3.2, l'utilisation d'un métamodèle doit permettre un minimum de compréhension de tous les fichiers de configuration. Avant de présenter les différentes technologies nous permettant d'utiliser un métamodèle, nous allons essayer de préciser l'intérêt d'un métamodèle dans le cadre du *WebTree*. Nous présenterons ensuite les quelques métamodèles que nous avons envisagé d'utiliser et nous terminerons par expliquer les raisons déterminant le choix du langage de représentation d'un métamodèle.

3.3.1 Intérêt d'un métamodèle

Afin de montrer l'intérêt d'un métamodèle, nous allons essayer de décrire le fonctionnement de l'application qui serait capable de réaliser l'édition de tous les fichiers de confi-

guration.

Prenons un fichier de configuration représentant un *Package* du *WebTree*. Ce fichier de configuration doit donc contenir la description de tous les *ClassTypes*, *NodeTypes*, *Policys* et actions. La description de chacun de ces éléments doit quant à elle correspondre au point de vue sémantique aux modèles que nous avons présentés au chapitre 2 (cf. 2.1.2, page 36). On pourrait penser qu'il suffit de définir une *DTD* décrivant complètement le concept de *Package*. En lisant la *DTD*, notre application pourrait "comprendre" le concept de *Package* et donc le contenu du fichier de configuration. Malheureusement, la fonction première de la *DTD* est de définir la structure d'un document. Or, de nombreuses notions apparaissant dans un modèle orienté objets comme celui d'un *Package* ne sont pas représentables en utilisant les seuls éléments de définition d'un document. Comme nous l'avons vu au point 2.1.2.5, un *NodeType* possède un *parentNodeType*. D'après le modèle l'objet référencé par cette association est aussi un *NodeType*. Or, le mécanisme de référence qui existe en *XML* permet de référencer un objet ayant un certain identifiant mais sans tenir compte du type de cet objet. Ainsi, si par hasard, un *ClassType* et un *NodeType* d'un même *Package* possèdent le même nom (ce qui est tout à fait possible d'après notre modèle), il n'est pas possible en *XML* de préciser si on référence le *ClassType* ou le *NodeType*. On voit donc que le concept de *DTD* est certainement suffisant pour décrire la structure d'un document mais en aucun cas pour représenter un modèle orienté objets.

Admettons maintenant que plutôt que de représenter le concept de *Package*, la *DTD* décrive un ensemble d'objets nous permettant de représenter les différentes notions nécessaires à la représentation du concept de *Package* et d'instances de ce concept. Cette *DTD* serait la même pour tous les fichiers de configuration. Par contre, nous aurions besoin d'un fichier supplémentaire où serait décrit le concept de *Package*. L'ensemble des objets décrits dans la *DTD* constituerait notre métamodèle. Au chargement du fichier de configuration de notre *Package*, l'application commencerait par lire le fichier décrivant le concept de *Package*. Elle pourrait ensuite interpréter le contenu du fichier contenant la description des instances des *ClassTypes*, *NodeTypes*, *Policys* et actions. Pour reprendre l'exemple ci-dessus, l'application saurait maintenant que l'objet référencé par le *parentNodeType* est bien un *NodeType*.

Nous avons décidé d'utiliser ce principe pour le *WebTree*. Ainsi, grâce au métamodèle que nous avons choisi, le *WebTree* est suffisamment générique pour pouvoir éditer tous les fichiers de configuration d'*OpenMaster* pour autant que l'on fournisse le fichier décrivant le modèle d'objets du fichier de configuration. En lisant le fichier de description des concepts, le *WebTree* génère automatiquement un *Package* complet d'édition pour ce concept. Les premiers modèles que nous avons représentés en utilisant le métamodèle sont tout simplement ceux représentant les différents concepts du *WebTree*. Le *WebTree* a ainsi généré son propre *Package* d'édition. Une fois celui-ci généré, il est possible de l'éditer (au sein même du *WebTree*) afin de l'enrichir pour, par exemple, utiliser un éditeur de noeuds en partie droite du *WebTop* (cf. 2.2.4).

Maintenant que nous avons montré l'intérêt d'utiliser un métamodèle pour le *WebTree*, nous allons présenter les différents métamodèles pouvant être utilisés avec *XML* que nous avons examinés.

3.3.2 *RDF*

Le *Resource Description Framework* ou *RDF* a été développé par un groupe de travail du *W3C*. Il s'agit d'une base destinée à fournir une interopérabilité entre des applications qui s'échangent sur le *WWW* des informations "compréhensibles" par un ordinateur. *RDF* fournit donc un ensemble de fonctionnalités afin de permettre le traitement automatique de ressources *WWW*. La syntaxe du *RDF* utilise le *XML*. En effet, un des buts de *RDF* est de rendre possible la spécification d'une sémantique pour des données tout en se basant sur le *XML*. Le *RDF* et le *XML* sont donc complémentaires : le *RDF* est un métamodèle et fournit accessoirement différentes fonctionnalités afin de pouvoir transporter et enregistrer les données. A cette fin, le *RDF* repose sur *XML*. Il faut toutefois noter que le *RDF* pourrait être utilisé avec une autre syntaxe que *XML*.

Le modèle de données de base pour représenter une expression *RDF* est constitué de trois types d'objets :

Les ressources : tout ce qui est décrit par une expression *RDF* est appelé une *ressource*. Il peut s'agir d'une page *WWW*, d'un document *XML* ou encore d'un élément d'un document *HTML* ou *XML*. Dans notre cas, il s'agirait d'un objet (*Node*, *ClassType*, *Package*, etc.) de notre application ;

Les propriétés : une *propriété* est une caractéristique, un attribut ou encore une relation utilisée pour décrire une *ressource*. Chaque propriété possède une signification particulière. Elle définit également un ensemble de valeurs possibles, le type de ressources qu'elle décrit et ses relations avec les autres propriétés ;

Les énoncés : une *ressource* spécifique, une *propriété* et la valeur de cette propriété constituent un *énoncé RDF*. Ces trois parties de l'énoncé sont appelées respectivement le *sujet*, le *prédicat* et l'*objet*. L'*objet* d'un *énoncé* peut être une référence vers une autre *ressource* ou une donnée.

Un document RDF est également constitué de deux parties : une partie définissant le schéma de données et une pour représenter les instances. La première partie permet de définir les différents types de ressources, ainsi que les différentes propriétés existantes. Elle permet de représenter n'importe quel modèle de données orienté objets. Néanmoins, il existe une différence importante au niveau de la façon de représenter le modèle. Alors que dans un modèle orienté objets traditionnel, une classe est définie en décrivant les propriétés

qu'elle peut avoir, un schéma *RDF* définit les propriétés en exprimant les différentes classes auxquelles elles peuvent être associées. Ainsi, en *RDF* on définira une propriété "auteur" comme pouvant être utilisée avec la classe "Livre" et ayant comme domaine de valeurs "littéraux". En orienté objets classique, on aurait défini la classe "Livre" avec une propriété "auteur" ayant comme domaine de valeurs "littéraux".

Nous ne décrirons pas la syntaxe du *RDF*. Néanmoins, elle nécessite une remarque importante : il est possible d'écrire un même énoncé de plusieurs manières. En effet, outre la syntaxe complète, il existe une syntaxe abrégée qui peut être utilisée afin par exemple de réduire la taille des documents. De plus, dans certains cas, il est également possible d'omettre certains attributs même si parfois cela peut être source d'ambiguïté.

3.3.3 *XML-Data*

Tout comme *RDF*, *XML-Data* permet de décrire des classes dans un document *XML*. Une différence importante est que, contrairement à *RDF* qui utilise *XML*, *XML-Data* consiste en un certain nombre d'ajouts à la syntaxe *XML* afin de pouvoir définir un modèle d'objets dans un document. De plus, il s'agit ici d'une modélisation relativement standard des objets. Ce modèle peut autant être défini en début de document que dans un document externe.

Bien que très complète, cette note [XML98b] du *W3C* émise au mois de janvier 1998 n'a plus évolué, la technologie la plus prisée au sein du *W3C* et des développeurs étant le *RDF*.

3.3.4 *CIM*

Contrairement aux deux précédents métamodèles, le *Common Information Model* (ou *CIM*) a été réalisé non pas au *W3C* mais bien au *DMTF*[3]. Il s'agit d'une organisation regroupant différentes sociétés. Elle s'intéresse principalement au développement, à l'adoption et à l'unification de standards de gestion et d'initiatives dans les environnements Internet, d'entreprises et de bureau. Travaillant avec des entreprises à la pointe au niveau technologique ainsi qu'avec des groupes de standardisation, le *DMTF* permet une meilleure intégration avec un profit maximum et au moindre coût des différentes solutions d'administration.

CIM constitue une approche à l'administration de systèmes et de réseaux qui applique les techniques de conception et de structuration de l'orienté objets. Cette approche utilise un formalisme de modélisation supportant le développement coopératif de schémas orientés objets au travers de différentes organisations. Ce métamodèle basé sur *UML* permet la

[3]DMTF : anciennement *Desktop Management Task Force* rebaptisé *Distributed Management Task Force* depuis le 10 mai 1999.

représentation d'éléments génériques tels que des classes, des méthodes, des associations, etc. devant être présentés clairement à des applications d'administration. Il présente donc l'avantage de pouvoir générer assez simplement l'ensemble des objets *CIM* correspondant à un schéma *UML* donné. Outre le métamodèle de base, *CIM* comporte un certain nombre d'extensions destinées à modéliser des objets spécifiques à certaines technologies ou environnements comme Windows ou Unix.

Au départ, *CIM* utilisait uniquement une notation propre afin de représenter textuellement le métamodèle. Cette notation est appelée *MOF*[4]. Néanmoins, dans le courant du mois de septembre 1998, le *DMTF* publia un ensemble de spécifications ainsi qu'une *DTD XML* afin de pouvoir utiliser *XML* pour représenter les objets *CIM*. Afin d'être très compréhensibles à la lecture, les différents *tags* utilisés dans cette représentation sont parfois très longs. Par exemple, la balise représentant une instance d'une association est "ASSOCIATION.INSTANCE". Ainsi, pour chaque instance d'une association (et comme nous l'avons vu sur les schémas *UML* présentés au point 2.1.2, elles sont nombreuses), cette balise sera utilisée à deux reprises : pour ouvrir et pour fermer la section représentant l'association. On peut donc considérer un document *XML* représentant un métamodèle *CIM* comme étant très verbeux.

3.3.5 Le métamodèle choisi

Un point crucial dans notre choix fut la complexité de mettre en oeuvre un des différents modèles. Contrairement à *XML*, il n'existe en effet aucun programme permettant d'interpréter les différentes syntaxes de ces métamodèles. A moins de générer un document *DOM* puis de parcourir celui-ci afin de repérer les élémens spécifiques au métamodèle, l'application doit elle-même interpréter les différents éléments de syntaxe du métamodèle choisi. Or, comme nous l'avons vu au point 3.2.4, nous avons écarté l'utilisation de *DOM*. Le *WebTree* doit donc lui-même comprendre toute la syntaxe (ou du moins sa majeure partie) du métamodèle choisi.

Comme nous avons pu le voir dans la partie 3.3.3 concernant *XML-Data*, ce métamodèle réalisé au début de l'année 1998 n'a plus beaucoup évolué, le *RDF* lui étant généralement préféré car offrant plus de possibilités et d'ouvertures. Nous avons donc nous aussi écarté cette technologie. Il ne nous restait donc plus qu'à choisir entre *RDF* et *CIM*.

Nous avons vu que *CIM* est excessivement verbeux. Cela pourrait-il être contraignant pour notre application ? Cette verbosité peut être très gênante si quelqu'un utilise un éditeur de textes pour modifier les documents *XML*. Or, nous avons vu que le but de notre application était justement de permettre à tout un chacun d'éditer assez simplement les différents fichiers de configuration sans pour autant devoir connaître la syntaxe utilisée dans ces fichiers. Il est donc fort improbable qu'un utilisateur édite ces fichiers manuellement.

[4]MOF : *Management Object Format.*

La taille des fichiers pourrait également être gênante pour les transferts entre le serveur *OpenMaster* où sont stockés ces fichiers et la machine client qui exécute l'applet *WebTop*. Or, ces fichiers sont des documents textes. Ceux-ci se compressent donc très bien avec des utilitaires de compression tels que *zip*. De plus, il existe un ensemble de classes Java permettant de compresser et de décompresser des fichiers en utilisant cette technique de compression. Si la taille des fichiers venait à ralentir excessivement le chargement des fichiers de configuration, il serait donc très simple d'utiliser la compression afin de pallier ce problème. Nous considérerons donc que la taille des fichiers ne constitue pas un frein à l'utilisation de *CIM*.

Essayons maintenant de voir si l'une ou l'autre des technologies est plus facile à mettre en oeuvre. *CIM* étant basé sur *UML*, son utilisation nous permettrait de transformer plus aisément nos schémas vers le métamodèle. En effet, le *RDF* utilisant une représentation orientée objets très différente, cette transformation serait légèrement plus complexe. De plus, comme nous l'avons vu dans la partie 3.3.2 consacrée à *RDF*, il est possible d'écrire un même objet de plusieurs façons (notations abrégées, etc.). Notre application devra donc être capable de comprendre ces différentes notations, ce qui complexifie fortement l'utilisation de cette technologie. Enfin, lors de notre première étude dans le courant du mois de septembre 1998, le *RDF* n'était pas encore finalisé. Ses spécifications complètes datent en effet du mois d'octobre 1998.

Les différents points que nous venons d'exposer nous ont donc conduit à utiliser *CIM* comme métamodèle. Nous avons ainsi rédigé deux fichiers *CIM/XML* contenant respectivement la structure des *Views* et celle des *Packages*. Ces fichiers étant réalisés, nous avons ajouté deux nouvelles fonctionnalités au *WebTree*. Tout d'abord, celui-ci peut interpréter les fichiers décrivant, grâce à *CIM/XML*, un modèle d'objets. Il génère alors un fichier représentant un *Package* d'édition pour les fichiers d'instances d'objets *CIM/XML*. La deuxième fonctionnalité est bien entendu le fait de pouvoir éditer un fichier d'instances d'objets *CIM/XML* en utilisant le *Package* d'édition correspondant.

3.4 Conclusion

Nous voici arrivé à la conclusion de ce chapitre durant lequel nous nous sommes intéressé d'un peu plus près aux différentes technologies que nous avons abordées durant le développement du *WebTree*. Nous avons ainsi présenté les nouveaux composants graphiques de Java : Swing.

La deuxième partie de ce chapitre fut alors consacrée à décrire *XML* et les différentes *API*'s qui permettent à une application de travailler assez simplement avec des documents *XML*.

Enfin, au travers des parties consacrées à *XML* et aux métamodèles, nous avons montré comment nous avons répondu au besoin d'avoir un format de fichier de configuration unique

et comment nous avons fait du *WebTree* une application générique capable non seulement de représenter les différents domaines d'objets d'*AGOV* mais également d'éditer tous les fichiers de configuration qui utilisent ce format.

Nous allons maintenant décrire plus précisément quelques aspects du développement du *WebTree* répondant aux derniers besoins exprimés au chapitre 1. Nous montrerons également dans ce dernier chapitre l'environnement de travail dans lequel nous étions ainsi que les différentes contraintes posées par le travail en équipe et les délais de livraison des développements effectués.

Chapitre 4

Le développement

Nous voici arrivé au dernier chapitre de ce mémoire consacré à tout ce qui concerne le développement proprement dit du *WebTree*. Contrairement aux chapitres précédents, celui-ci sera moins théorique, plus concret et plus orienté vers un aspect souvent écarté jusqu'à présent : le codage de l'application.

Nous commencerons par présenter l'environnement de travail qui était mis à notre disposition pour l'analyse, le codage ou encore les tests de notre application. Nous essayerons également dans cette partie de montrer l'existence de nombreuses interactions entre l'équipe de développement où nous étions intégré et les autres équipes existant au sein d'*OpenMaster*.

La deuxième partie de ce chapitre s'attachera quant à elle à essayer de montrer la structure générale de l'application non plus vis-à-vis de l'interface mais plutôt au point de vue du code du *WebTree*. Nous montrerons ainsi comment nous sommes passé des différents concepts présentés au chapitre 2 au code de l'application. Cette partie montrera également comment nous avons répondu aux deux derniers besoins, à savoir une *API* d'édition du *WebTree* destinée aux autres applications d'*OpenMaster* et l'internationalisation du *WebTree*.

Enfin, la dernière partie de ce chapitre tentera de montrer tous les problèmes rencontrés lors du développement. Nous verrons ainsi qu'il fut bien difficile de planifier complètement le développement, que cela soit dû à un changement au niveau des desiderata, à l'apparition de nouveaux besoins ou encore à des difficultés de synchronisation avec les différents développements en cours dans *OpenMaster*.

4.1 L'environnement

Cette partie va être consacrée à essayer de décrire l'environnement de travail dans lequel nous étions durant notre stage. Nous allons examiner cet environnement de travail selon deux points de vue : les ressources humaines et le matériel / logiciel informatique.

4.1.1 Les ressources humaines

Tout d'abord, bien que nous étions au sein de l'équipe *Monitor* chargée de développer l'application du même nom au sein d'*OpenMaster*, nous ne travaillions nullement sur ce projet. Tout notre temps était consacré au développement du *WebTree* dans le cadre du projet *AGOV*. Nous avions ainsi de nombreux contacts avec le responsable du projet *AGOV* afin d'étudier les différents besoins et les différents développements nécessaires pour y répondre. Certains développements n'étant possibles que moyennant des ajouts de fonctionnalités au sein d'autres applications *OpenMaster* (souvent le *WebTop*, vu les liens étroits unissant ce dernier et le *WebTree*), nous entretenions également des contacts réguliers avec les autres équipes de développement. Un certain nombre de réunions furent également consacrées à l'étude des besoins d'autres projets vis-à-vis du *WebTree* (cf. 1.2.2, page 26). Enfin, nous avions également de nombreuses entrevues avec le responsable de l'équipe *Monitor* afin soit de rendre compte de l'état d'avancement de l'analyse ou du développement, soit de discuter de certains choix technologiques.

4.1.2 Le matériel / logiciel informatique

Au point de vue matériel, nous disposions, tout comme les deux autres développeurs du bureau où nous étions installé, d'un PowerPC avec le système d'exploitation IBM AIX 4.3. Un serveur *OpenMaster* était installé sur chacun de ces trois ordinateurs. Nous avions également à notre disposition deux ordinateurs avec Windows NT Workstation 4.0 afin de pouvoir exécuter l'applet *WebTop* au sein du navigateur Internet Explorer et donc de pouvoir tester nos développements respectifs.

Le choix d'un outil de modélisation *UML* étant en cours, nous avons utilisé, durant l'analyse, la version de démonstration du logiciel Rational Rose afin de réaliser les diagrammes *UML*. Plus tard, ce fut l'application Objecteering qui fut utilisée afin, notamment, de réaliser les diagrammes *UML* relatifs à la gestion des rôles.

La *JDK* utilisée fut la version 1.1.5, dernière version disponible sur AIX 4.3 durant notre stage. Au niveau de Swing, nous avons commencé certains tests avec la version beta avant de passer à la version 1.1.

Enfin, nous avons également utilisé les différents outils de développement sous Unix tels que make[1], imake[2], *RCS*[3], *CVS*[4], etc. afin d'automatiser les compilations et de gérer les différentes versions du *WebTree* et les autres composants développés au sein de l'équipe.

[1]make : utilitaire Unix destiné à l'exécution de fichier "Makefile" c'est-à-dire de fichiers contenant un ensemble de règles d'exécution de commandes Unix. L'utilisation de ces fichiers permet un gain de temps non négligeable lors de la compilation de gros programmes.

[2]imake : générateur de fichiers *Makefile* destinés à l'application *make* afin de rendre plus aisé le développement d'applications pour plusieurs systèmes d'exploitation.

[3]RCS : *Revision Control System.*

[4]CVS : *Concurrent Versions System.*

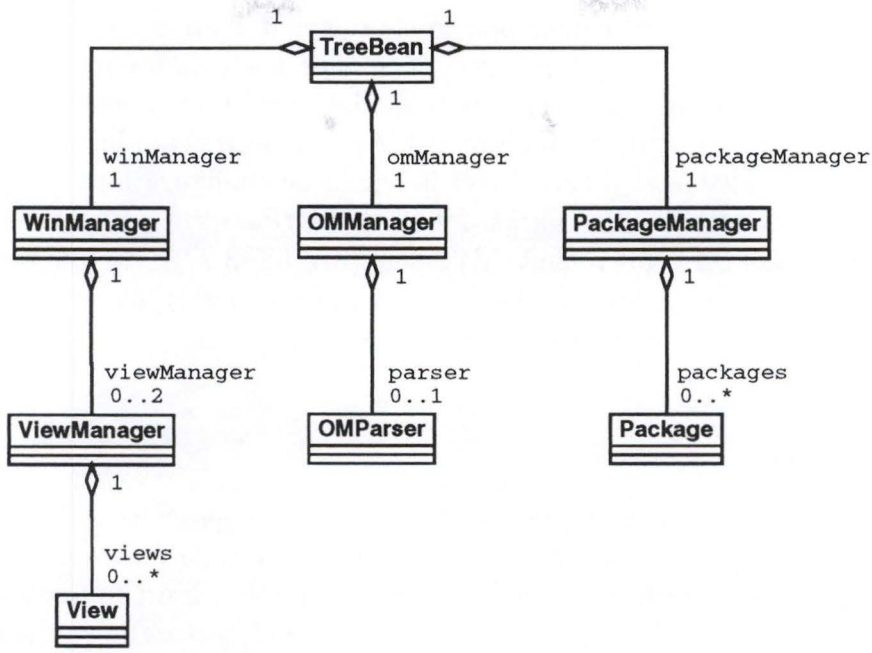


FIG. 4.1 – Les objets principaux du *WebTree*

4.2 De l'analyse au code

Examinons maintenant un peu plus en profondeur le développement. Nous allons commencer par présenter l'organisation des principales classes du *WebTree*. Nous verrons ensuite comment nous avons pu générer une partie du code ainsi que certains fichiers de configuration. Nous pourrons alors montrer comment nous avons répondu aux deux derniers besoins, à savoir une *API* d'édition du *WebTree* destinée aux autres applications d'*OpenMaster* et l'internationalisation du *WebTree*.

4.2.1 L'organisation du *WebTree*

La figure 4.1 présente les principales classes du *WebTree*. La première de ces classes, *TreeBean*, représente le *bean* Java qui sera créé par le *WebTop*. Elle contient trois autres classes (*OMManager*, *PackageManager* et *WinManager*) qu'elle utilisera afin de répondre aux demandes de l'utilisateur. Examinons les rôles joués par chacune de ces classes.

OMManager

La classe *OMManager* a pour but de charger en mémoire les différents composants d'un fichier de configuration. Outre la classe *OMParser* représentée sur la figure 4.1, elle a à sa disposition un ensemble d'objets représentant les différents éléments de définition de classes tels qu'existant dans le métamodèle *CIM*.

Lorsque la classe *TreeBean* demande le chargement d'un fichier de configuration, la classe *OMManager* commence par charger (si ce n'est pas déjà fait) le fichier de définition de classes afin de représenter en mémoire le modèle des objets contenu dans le fichier de configuration. Ce modèle peut ensuite être utilisé par les autres modules du *WebTree* afin de créer un *Package* d'édition et d'enregistrer le fichier de configuration. Le fichier contenant les instances d'objets est ensuite chargé. La classe *OMParser* a pour but de répondre aux différents événements *SAX* provenant du parser *XML* (cf. 3.2.2). En fonction des différents événements reçus, elle crée les différents objets internes au *WebTree* et les transmet au module qui demande le chargement du fichier.

PackageManager

La classe *PackageManager* a quant à elle pour but de gérer les différents *Packages* en mémoire. Tout module du *WebTree* désirant accéder à un *Package* demande celui-ci à la classe *PackageManager*. Si le *Package* est déjà en mémoire, il est fourni au module. Dans le cas contraire, la classe *PackageManager* demande à *TreeBean* le chargement du fichier de configuration correspondant au *Package*. Ce chargement est alors effectué grâce à la classe *OMManager*, les différents objets créés par la classe *OMParser* lors du chargement constituant le *Package*. Celui-ci peut maintenant être transmis au module qui le demandait. Il est bien entendu aussi possible de demander la création d'un nouveau *Package*, cette création pouvant éventuellement se baser sur un modèle d'objets contenu par la classe *OMManager* afin de réaliser un *Package* d'édition pour celui-ci.

WinManager

Enfin, la classe *WinManager* est responsable du type d'affichage : une ou deux vues. Elle utilise à cette fin les instances de la classe *ViewManager*. Les différentes instances de cette classe se partagent l'accès aux différentes instances de la classe *View*. Ainsi, une seule instance d'une vue est présente en mémoire même si plusieurs instances de la classe *ViewManager* existent. Néanmoins, plusieurs instances de la classe *ViewManager* ne peuvent accéder à une même instance de la classe *View* en même temps.

Afin de charger une vue, l'objet *TreeBean* s'adresse à l'objet *WinManager*. Si besoin est, celui-ci crée un objet *ViewManager* et lui demande de charger la vue désirée. Ce chargement est alors effectué, comme pour le chargement d'un *Package*, en faisant appel à la classe *OMManager*.

L'objet *TreeBean* peut également demander à l'objet *WinManager* de changer de mode d'affichage, c'est-à-dire de passer du type "Une vue" au type "Deux vues".

Enfin, pour la fermeture ou l'enregistrement d'une vue, l'objet *TreeBean* passe également par la classe *WinManager* qui relaie la demande à l'objet *ViewManager*, lui-même

la transmettant à la *View*. Dans le cas d'une sauvegarde, celle-ci peut utiliser le modèle d'objets conservé par l'objet *OMManager*.

4.2.2 La génération du code

Comme nous l'avons dit durant l'introduction de ce chapitre, différents outils de modélisation *UML* furent utilisés, soit par nous, soit au sein d'autres équipes. Ces outils offrent généralement la possibilité de générer le code correspondant aux différents objets modélisés, les langages le plus souvent supportés à l'heure actuelle étant le C++ et le Java. Néanmoins, certains logiciels, et notamment Objecteering, offrent la possibilité de définir la syntaxe d'un autre langage afin de pouvoir générer le code dans ce dernier. Ceci consiste tout simplement à définir pour tous les objets et pour toutes les propriétés pouvant être appliquées sur ces objets la syntaxe correspondante dans le nouveau langage.

C'est ainsi qu'à partir des différents diagrammes *UML* des concepts internes au *WebTree*, nous avons pu générer une grande partie des fichiers Java.

De même, la personne ayant réalisé le modèle des objets pour la gestion des rôles au sein d'*OpenMaster* ajouta à Objecteering la possibilité de générer le fichier *CIM/XML* de définition de classes. Cette possibilité fut alors utilisée sur ce modèle afin d'avoir un fichier de définition de classes compréhensible par le *WebTree* qui put l'utiliser afin de créer un *Package* d'édition pour les fichiers de configuration des rôles *OpenMaster*.

4.2.3 La réponse aux derniers besoins

En nous basant tout d'abord sur le modèle présenté puis sur les possibilités de Java, nous allons maintenant expliquer comment nous avons répondu aux deux derniers besoins. Nous commencerons par le besoin de fournir aux différentes applications d'*OpenMaster* une *API* permettant de charger, fermer, éditer, etc. une vue du *WebTree*.

4.2.3.1 Une *API* d'édition

Comme nous l'avons vu dans le modèle présenté au point 4.2.1, les différentes classes du *WebTree* s'utilisent les unes les autres. Néanmoins, elles réalisent toutes leurs demandes en s'adressant à l'objet *TreeBean*. Cette classe offre donc déjà une *API* aux différentes classes du *WebTree* pour tout ce qui concerne la gestion des *Package*s, la gestion des fichiers de configuration et la gestion des vues et de leur affichage. Il ne manquait donc plus qu'un ensemble de fonctionnalités permettant d'éditer un noeud. Or, ces fonctionnalités étaient également présentes dans l'objet *TreeBean*, mais dans un autre but. En effet, lorsque nous avons décrit les besoins concernant les actions (cf. 1.2.1.3), nous avons vu qu'un ensemble d'actions internes au *WebTree* étaient nécessaires afin d'ouvrir ou fermer un *bean* depuis

l'arbre, d'ouvrir une fenêtre du navigateur sur une certaine *URL*, ou encore de pouvoir modifier les caractéristiques d'un noeud. De même, dans la partie consacrée au concept de *Policy* (cf. 2.1.2.6), nous avons parlé d'actions permettant de déplacer ou de dupliquer un noeud. Ces actions doivent également être internes au *WebTree*.

Toutes les fonctionnalités nécessaires pour fournir une *API* complète ont donc été ajoutées afin de respecter d'autres besoins que celui de l'*API*. Nous avons donc tout simplement décidé de créer une interface Java regroupant toutes ces méthodes, la classe *TreeBean* implémentant cette interface. Toutes les méthodes définies dans une interface doivent être de type *public*, c'est-à-dire qu'elles doivent pouvoir être appelées depuis n'importe quelle classe. Ainsi, toutes les applications d'*OpenMaster* peuvent utiliser les méthodes déclarées dans l'interface d'édition afin de charger, éditer, fermer, etc. une vue du *WebTree*. A l'exception de certains cas précis (et requis par le langage Java), les autres méthodes, que ce soit au sein de la classe *TreeBean* ou des autres classes de notre application, sont de type *package* ou *private*, c'est-à-dire qu'elles peuvent seulement être appelées soit depuis un objet du même *package* Java, soit depuis l'objet où elles sont définies.

Enfin, notons le fait que l'utilisation d'un canal unique pour réaliser les différentes fonctionnalités du *WebTree* nous a permis d'insérer aisément les fonctionnalités relatives à la gestion des rôles, fonctionnalités qui n'apparurent qu'à la fin de notre stage. En effet, une fois les appels nécessaires à la gestion des rôles insérés à quelques endroits bien déterminés au sein du *WebTree*, cette gestion était présente quelle que soit la manière d'utiliser le *WebTree* : au travers de l'interface utilisateur ou depuis une autre application en utilisant l'*API* d'édition.

4.2.3.2 L'internationalisation

Il nous fut également assez simple de répondre au besoin d'internationalisation. Java offre en effet un système permettant de gérer assez aisément la cohérence de la langue à utiliser en fonction de la langue du système où est exécutée l'applet ou l'application Java. C'est ce système qui est utilisé généralement dans *OpenMaster*.

Nous avons donc également utilisé ce système basé sur l'existence, pour chaque langue, d'un fichier de ressources qui associe une chaîne de caractères à un mot clé. Nous avons tout simplement considéré que toutes les valeurs de caractéristiques que l'utilisateur pouvait entrer durant l'édition d'une vue était un mot clé. Ainsi, lorsque la vue est utilisée en mode normal, nous faisons tout simplement appel aux fonctionnalités de Java pour avoir la valeur réelle, donc internationalisée, à afficher. Si aucune valeur n'est définie dans les fichiers d'internationalisation, le mot clé est affiché. Ceci permet aux différentes équipes d'*OpenMaster* de livrer les fichiers d'internationalisation correspondant aux vues qu'ils ont développées tout en laissant la possibilité au client de créer de nouvelles vues dans sa propre langue. Afin que le client ne doive pas se soucier de ce qu'il entre comme valeurs de caractéristiques, les valeurs entrées pour l'internationalisation respectent un format

permettant notamment d'identifier le fichier d'internationalisation à utiliser mais également de diminuer la probabilité d'un quelconque conflit avec une valeur réelle.

4.3 Les problèmes rencontrés

Nous allons maintenant nous orienter vers un autre aspect des développements réalisés durant notre stage : les problèmes rencontrés. Le premier point que nous allons aborder concerne la taille de l'application *OpenMaster*. Nous parlerons ensuite des problèmes causés par la difficulté de synchroniser des projets différents mais dont les développements dépendent parfois les uns des autres. Enfin, nous parlerons des contraintes de temps en général.

4.3.1 *OpenMaster*

Bien que nous n'ayons décrit qu'une toute petite partie d'*OpenMaster*, il est facile de se rendre compte de la taille de cette application. Un des premiers problèmes rencontrés fut de découvrir petit à petit un certain nombre des fonctionnalités et des possibilités d'*OpenMaster* afin d'être à même de bien comprendre les différents besoins dont nous avons parlé.

De même, le *WebTree* devant s'intégrer dans le *WebTop*, il fallut également apprendre à connaître les différentes *API* déjà réalisées au sein d'*OpenMaster* afin de pouvoir utiliser celles-ci correctement. Ainsi, même si nous n'avons que peu de notions de la manière dont le *WebTop* travaille, nous connaissons maintenant les différentes interactions possibles avec ce dernier. Il en est bien entendu de même pour ce qui concerne les autres applications ou services *OpenMaster* avec lesquels le *WebTree* doit communiquer.

4.3.2 La synchronisation des développements

Comme nous l'avons déjà souligné à plusieurs reprises, le *WebTree* doit interagir avec plusieurs applications et services *OpenMaster* afin de remplir convenablement sa fonction et donc de répondre aux différents besoins concernant *AGOV*. Or, certains besoins exprimés par *AGOV* nécessitaient des développements au sein d'autres applications et donc réalisés par d'autres équipes. Ces développements devaient alors s'intégrer parmi les autres développements en cours dans ces équipes. On comprendra aisément qu'à certains moments d'autres développements que nous qualifierons de plus prioritaires firent que nous fûmes bloqué dans nos propres développements.

Ce fut notamment le cas avec Swing. En effet, comme nous l'avons expliqué, pour qu'un composant Swing fonctionne correctement, il ne peut être inséré dans un composant non Swing. Or, au début des développements, toutes les applications d'*OpenMaster* étaient

réalisées en utilisant uniquement *AWT*. Le passage du *WebTop* d'*AWT* à Swing demandait un développement assez important en son sein. Or, un autre développement étant en cours, l'équipe de développement du *WebTop* ne put pas réaliser cette migration tout de suite. Afin de ne pas être bloqué trop longtemps nous avons donc décidé de commencer à développer le *WebTree* sous la forme d'une application et donc d'implémenter dans un premier temps toutes les fonctionnalités (par exemple la lecture et l'interprétation des fichiers *CIM/XML*) qui pouvaient exister sans la présence du *WebTop*. Une fois le *WebTop* passé en Swing, nous n'avons eu qu'à réaliser quelques changements minimes afin de transformer notre application en un *bean* destiné à être contenu par le *WebTop*. Nous aurions également pu commencer les développements en *AWT* puis passer en Swing par après mais cela aurait demandé des changements beaucoup plus importants et beaucoup plus coûteux en temps.

4.3.3 Les contraintes de temps

Le projet sur lequel nous avons travaillé s'intégrait dans une phase complète de développements au sein d'*OpenMaster*. Les différents développements sont bien entendu planifiés plusieurs mois à l'avance. On comprendra, vu la taille d'*OpenMaster*, que plusieurs versions de celui-ci soient en développement en même temps avec des échéances différentes.

En ce qui nous concernait, l'échéance était le mois de mars 1999, date à laquelle toutes les applications appartenant à la même version d'*OpenMaster* que nous devaient être prêtes. Or, notre stage se terminait fin janvier. Ceci explique en partie un grand nombre des problèmes de synchronisation entre les différentes équipes. Lorsque nous avons terminé notre stage, certains développements externes au *WebTree* n'avaient d'ailleurs pas encore été réalisés. Il ne nous fut donc pas possible de développer complètement l'application durant notre stage. C'est ainsi, par exemple, qu'au moment où notre stage s'est clôturé, le fonctionnement de l'animation n'était pas encore bien défini en ce qui concerne les interactions entre le *WebTree* et le système d'alarmes encore en développement au sein d'*OpenMaster*.

La fin de notre stage fut donc consacrée à expliquer les différents concepts développés dans ce mémoire ainsi que le code réalisé à la personne chargée de poursuivre les développements concernant le *WebTree*. Cette personne étant déjà chargée d'autres développements, certaines fonctionnalités prévues mais non encore développées au moment de notre départ furent donc reportées à une version suivante de l'application. De plus, *OpenMaster* étant en constante évolution, on comprendra également qu'il y ait eu depuis lors des modifications au niveau des besoins et que, par exemple, certains développements dont nous avons parlé ici ne doivent plus être réalisés et que d'autres soient apparus.

Enfin, nous avons également dû tenir compte, lors des différents choix technologiques, du temps nécessaire à l'implémentation de ces technologies. Ainsi, nous avons notamment vu en ce qui concernait le choix entre *CIM* et *RDF* qu'un des critères ayant motivé notre

choix pour *CIM* fut le fait qu'il était nettement moins coûteux en temps de réaliser une application capable d'interpréter la syntaxe *CIM* que de développer une application pouvant comprendre les différentes syntaxes du *RDF*.

4.4 Conclusion

Nous voici arrivé au terme de ce chapitre où nous avons passé en revue les différents éléments relatifs aux développements réalisés durant notre stage.

Nous avons ainsi présenté l'environnement de travail dans lequel nous étions, tant au niveau des personnes avec lesquelles nous travaillions qu'au niveau du matériel et des logiciels qui étaient à notre disposition.

Nous avons ensuite montré et expliqué l'architecture des principaux composants de l'application *WebTree*. Nous avons alors vu comment l'utilisation de logiciels de modélisation orientée objets a pu nous aider dans la génération non seulement d'une partie du code Java mais également de fichiers *CIM/XML* contenant les définitions de classes. Enfin, nous avons terminé cette deuxième partie en présentant notre réponse aux deux derniers besoins : l'*API* d'édition et l'internationalisation du *WebTree*.

Enfin, nous avons clôturé ce chapitre par la présentation de quelques problèmes survenus durant les développements. Nous avons commencé par souligner l'importance de l'application *OpenMaster* et donc la difficulté d'en apprendre le fonctionnement. Nous avons également vu comment il pouvait être parfois complexe de synchroniser parfaitement des développements réalisés dans des équipes distinctes qui travaillent sur plusieurs projets. Finalement, nous avons expliqué pourquoi, au moment de notre départ, l'application *WebTree* n'était pas encore terminée et comment les délais de développements ont pu jouer sur certains choix tant au niveau des fonctionnalités à implémenter qu'au niveau des technologies à utiliser.

Conclusion

Il ne nous reste maintenant plus qu'à tirer les conclusions de cette année passée d'une part en stage, d'autre part dans la rédaction de ce mémoire. Nous commencerons par passer en revue le déroulement de notre stage que nous commenterons afin de montrer ce que nous avons pu apprendre et ce que nous aurions aimer changer. Nous aborderons ensuite la partie consacrée au mémoire proprement dit.

Lorsque nous sommes arrivé, chez Bull au début du mois d'août 1998, nous avons tout d'abord dû nous intégrer au sein de l'équipe. Cela veut dire non seulement apprendre à connaître et à travailler avec les membres de l'équipe mais également découvrir tout un environnement de travail que nous ne connaissions que très peu. C'est ainsi que les premières semaines furent consacrées principalement à la configuration complète, avec l'aide des différents membres de l'équipe, de ce qui devint notre poste de travail. Nous découvrîmes également peu à peu l'application *OpenMaster* et ses nombreuses possibilités. Ce fut vraiment une période très enrichissante, tant au niveau des connaissances d'outils de développements sous Unix, qu'en ce qui concerne l'administration de parcs informatiques.

Le mois d'août étant, en France, assez synonyme de vacances, une partie importante des développeurs étaient absents et on peut donc dire que les développements en cours tournaient au ralenti. D'autant plus que certains développements se terminaient et que d'autres projets se mettaient en place. C'est ainsi que nous avons été affecté à la réalisation d'une nouvelle application pour le projet *AGOV* : le *WebTree*. Nous avons alors commencé l'analyse de l'application. Celle-ci fut en fait constituée tout d'abord d'une série de réunions avec les responsables du projet *AGOV*. Après chaque réunion, nous essayions de remettre au clair les différents désiderata qui étaient apparus. De cette façon, lors de la réunion suivante nous pouvions présenter une idée de la manière de répondre à ces besoins. Dans certains cas, il apparaissait que ce que nous avions pensé permettait réellement de répondre aux besoins, dans d'autres cas nous devions faire marche arrière afin d'essayer de reclarifier notre compréhension des besoins. Bien que les responsables du projet *AGOV* avaient une idée assez précise de ce qu'ils désiraient avoir comme application, il n'était pas toujours facile de pouvoir bien comprendre tel ou tel besoin. Il y avait plusieurs raisons à cela. Tout d'abord, bien que nous nous familiarisions de plus en plus avec notre environnement en général et avec l'application *OpenMaster* en particulier, certains concepts nous restaient parfois inconnus. Ensuite, même si l'on essaie d'être le plus précis possible dans la description de ce que l'on désire, il reste souvent un point ou l'autre que l'on a

oublié de mentionner et qui fait que la personne se forge une image légèrement erronée de ce que l'on a essayé d'exprimer. Ce n'est qu'à force de questions et de discussions que l'on peut affiner de plus en plus les différentes visions des interlocuteurs afin que l'application que l'on développe corresponde parfaitement aux besoins du commanditaire. A mesure que nous affinions les besoins et les fonctionnalités de l'application, de nouvelles idées apparaissaient, nous forçant parfois à revenir en arrière et à revoir certaines parties que nous pensions terminées. De même, certains desiderata émis par *AGOV* étaient incompatibles avec d'autres provenant de projets différents. Nous devions alors essayer de trouver un compromis permettant sinon de contenter tout le monde, du moins de ne pas entraver le bon déroulement de l'un ou l'autre projet. C'est ainsi que nous sommes finalement arrivé à connaître les différents besoins que nous avons décrits dans le premier chapitre et à concevoir les différents concepts énoncés au chapitre deux, concepts autour desquels s'articulent toute l'application *WebTree*.

Toute cette période d'analye fut également d'un très grand intérêt, non plus au niveau de l'apprentissage de nouveaux logiciels, mais plutôt au niveau de l'expérience acquise dans le domaine de l'ingénierie des besoins. Il existe néanmoins un fait qui nous a particulièrement marqué durant cette analyse : le fait de considérer l'utilisateur comme quelqu'un de compétent, qui "sait parfaitement ce qu'il fait" avec la conséquence qu'il peut supprimer (pour autant qu'il possède les privilèges requis) des éléments de *Package*s au risque de mettre certaines vues dans un état incohérent. Nous aurions aimé pouvoir ne fût-ce qu'avertir l'utilisateur qu'en effectuant cette action, il pourrait rendre impossible l'utilisation de telle ou telle vue.

Nous avons ensuite passé quelques semaines à examiner les différents métamodèles et à nous documenter sur ceux-ci afin d'essayer d'en choisir un qui coïncide le mieux possible non seulement avec nos besoins mais également avec nos délais de développement relativement courts. Durant cette période, nous avons également commencé à nous familiariser avec Swing et à essayer de cerner les différents problèmes que son utilisation allait engendrer.

Ici aussi, quelques remarques peuvent être faites. Tout d'abord, nous avons constaté que ce n'est pas forcément parce qu'une technologie est la meilleure, ou la plus en vogue, qu'elle est celle qui correspond le mieux à nos besoins. Beaucoup d'autres paramètres, tels que les coûts de mise en oeuvre ou encore les délais nécessaires à son utilisation, doivent entrer en ligne de compte afin de déterminer laquelle est la plus adaptée à ce que l'on désire réaliser. La deuxième remarque que nous ferons à ce niveau concerne Swing et les problèmes liés à son utilisation. Nous trouvons dommage qu'il n'y ait pas eu de planification à un niveau plus global au sein d'*OpenMaster* en ce qui concerne le passage de *AWT* vers Swing. Ceci nous aurait probablement permis de pouvoir passer directement à la phase de développement du *bean WebTree* plutôt que d'avoir passé un certain temps à déceler les incompatibilités entre *AWT* et Swing.

Les différents besoins étant exprimés et les technologies à utiliser étant sélectionnées, nous sommes alors passé à la phase de développement proprement dit de l'application. De

par notre formation et mis à part un approfondissement de nos connaissances du langage Java, ce fut probablement la période qui fut la moins enrichissante au niveau des nouvelles connaissances apprises. C'est probablement dans cette partie que se situe également un de nos plus grands regrets concernant le *WebTree* : ne pas avoir eu la possibilité d'en terminer le développement, notamment à cause des différents problèmes liés à Swing mais également à cause des deux mois séparant la date à laquelle l'application devait normalement être terminée et la date à laquelle notre stage se clôturait.

Enfin, avant de conclure cette partie consacrée au stage, il nous semble opportun de préciser qu'au moment de la réalisation de l'application, un certain nombre de technologies évoquées n'étaient pas encore complètement approuvées par des organismes de normalisation comme le *W3C*. De même, nous avons parfois dû utiliser certains outils qui étaient eux-mêmes toujours en développement. Vu la rapidité d'évolution des technologies actuelles, il est probable que bon nombre de ces outils et technologies auront été modifiés depuis, que de nouveaux auront vu le jour. Il se pourrait que si nous devions, aujourd'hui, sélectionner les meilleures solutions pour répondre aux besoins de développements du *WebTree*, certaines seraient différentes de celles retenues durant notre stage.

Nous allons maintenant nous tourner pour quelques instants vers la rédaction de ce travail. Le premier point à mentionner est que tout au long de cette rédaction, nous avons essayé de rendre compte de la manière la plus compréhensible possible de tout ce qui avait trait au développement du *WebTree*. Alors que certaines parties furent relativement aisées à décrire, d'autres nécessitèrent beaucoup plus de travail, soit tout simplement à cause de la complexité de celles-ci, soit une nouvelle fois à cause de la difficulté qu'il peut y avoir à exprimer complètement et clairement quelque chose qui nous paraît très clair.

Pour conclure, nous dirons simplement que ce stage et ce mémoire nous ont permis non seulement de mettre en pratique un grand nombre de connaissances acquises au cours de ces dernières années, mais également d'en acquérir de nouvelles. De plus, et ce serait sans doute un des points les plus enrichissants de cette expérience, nous avons pu nous familiariser avec le monde du travail et ses contraintes, et ainsi nous rendre compte de la vanité des connaissances acquises tout au long de nos études tant qu'elles ne sont pas mises en pratique.

Table des sigles

AGOV	Application GOVernor
API	Application Programming Interface
AWT	Abstract Windowing Toolkit
CIM	Common Information Model
CMIP	Common Management Information Protocol
CVS	Concurrent Versions System
DTD	Document Type Definition
DMTF	Anciennement "Desktop Management Task Force" rebaptisé "Distributed Management Task Force" depuis le 10 mai 1999
DOM	Document Object Model
DSAC	Distributed Systems Administration and Control
ERP	Enterprise Resource Planning
HTML	HyperText Markup Language
IETF	Internet Engineering Task Force
imake	Générateur de fichiers "Makefile" destinés à l'application "make" afin de rendre plus aisé le développement d'applications pour plusieurs systèmes d'exploitation
ISO	International Standards Organization
JDK	Java Development Kit
JFC	Java Foundation Classes
JobScheduler	Application de planification d'exécution de travaux pour Oracle
make	Utilitaire Unix destiné à l'exécution de fichier "Makefile"
Makefile	Fichier contenant un ensemble de règles d'exécution de commandes Unix. L'utilisation de ces fichiers permet un gain de temps non négligeable lors de la compilation de gros programmes

Middleware	Selon [OHE98], ensemble des logiciels nécessaires à l'interaction entre un client et un serveur
MOF	Management Object Format
RCS	Revision Control System
RDF	Resource Definition Framework
SAP	Logiciel intégré de gestion financière, des ressources humaines et du marketing
SAX	Simple *API* for *XML*
SGML	Standard Generalized Markup Language
SNA	Systems Network Architecture
SNMP	Simple Network Management Protocol
Swing	Ensemble d'outils et de composants graphiques pour Java
TP	Transaction Processing
TUXEDO	Application transactionnelle
UML	Unified Modeling Language
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WWW	World Wide Web
X11	Système de fenêtrage pour les systèmes d'exploitation de type Unix
XML	Extensible Markup Language

Bibliographie

[CIM98a] Common Information Model (CIM) Specification, mars 1998. http ://www.dtmf.org/.

[CIM98b] Specification for the representation of CIM in XML, septembre 1998. http ://www.dtmf.org/.

[DOM98] Document Object Model (DOM) Level 1 Specification Version 1.0, octobre 1998. http ://www.w3.org/TR/1998/REC-DOM-Level-1-19981001.

[Eng97] Robert Englander. *Developing Java Beans.* O'Reilly, 1997.

[Fla97] David Flanagan. *Java in a Nutshell, Second Edition.* O'Reilly, mai 1997.

[OHE98] Robert Orfali, Dan Harkey, and Jeri Edwards. *Client / serveur Guide de Survie - 2ème édition.* Vuibert, juin 1998.

[RDF98a] Resource Description Framework (RDF) Model and Syntax Specification, octobre 1998. http ://www.w3.org/TR/1998/10/WD-rdf-syntax-19981008.

[RDF98b] Resource Description Framework (RDF) Schema Specification, octobre 1998. http ://www.w3.org/TR/1998/10/WD-rdf-schema-19981030.

[SGM] A Gentle Introduction to SGML. http ://sable.ox.ac.uk/ota/teip3sg/.

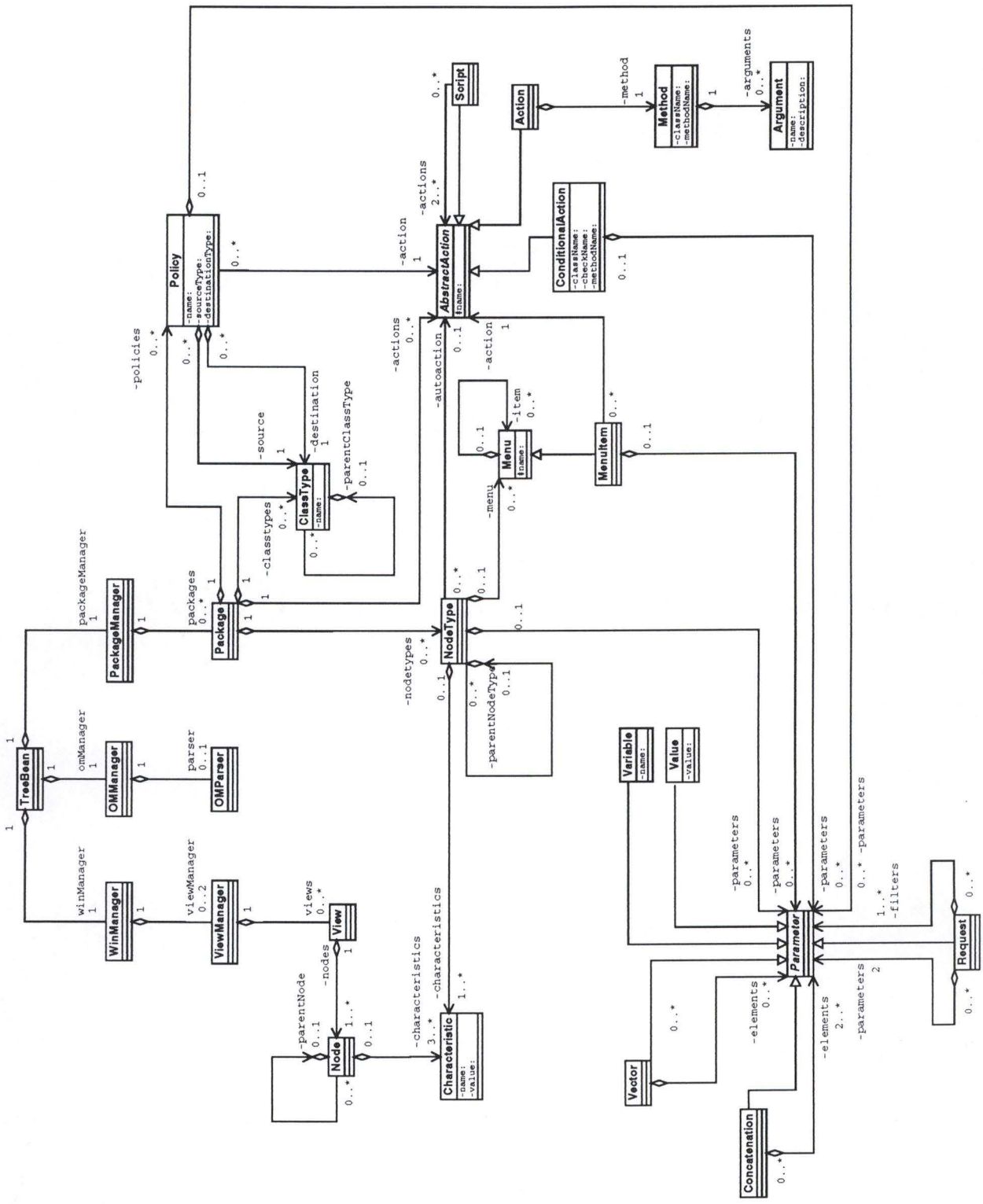
[UML97a] UML Semantics version 1.1, septembre 1997. http ://www.rational.com/uml.

[UML97b] UML Summary version 1.1, septembre 1997. http ://www.rational.com/uml.

[XML98a] Extensible Markup Language (XML) 1.0 W3C Recommendation, février 1998. http ://www.w3.org/TR/1998/REC-xml-19980210.

[XML98b] XML-Data, janvier 1998. http ://www.w3.org/TR/1998/NOTE-XML-data-0105.

Annexe : l'architecture du *WebTree*


TreeBean
OMManager
OMParser
WinManager
ViewManager
View
Node
PackageManager
Package
ClassType
Policy
NodeType
AbstractAction
Script
Action
Method
Argument
ConditionalAction
Menu
MenuItem
Characteristic
Variable
Value
Parameter
Vector
Concatenation
Request
-name:
-sourceType:
-destinationType:
-checkName:
-methodName:
-description:
-fname:
-tname:
-value:
packageManager
omManager
parser
winManager
viewManager
views
-nodes
-parentNode
-characteristics
packages
-classtypes
-source
-destination
-parentClassType
-policies
-action
-actions
-autoaction
-item
-menu
-nodetypes
-parentNodeType
-method
-arguments
-parameters
-elements
-filters